SW-WAL: Leveraging Address Remapping of SSDs to Achieve Single-Write Write-Ahead Logging

Qiulin Wu, You Zhou*, Fei Wu, Ke Wang, Hao Lv, Jiguang Wan, Changsheng Xie
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System
Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China
School of Computer Science & Technology, Huazhong University of Science & Technology
Email:{qiulin_wu, zhouyou2*, wufei, wangke25, lvhao, jgwan, cs_xie}@hust.edu.cn, *Corresponding Author

***Abstract*—Write-ahead logging (WAL) has been widely used to provide transactional atomicity in databases, such as SQLite and MySQL/InnoDB. However, the WAL introduces duplicate writes, where changes are recorded in the WAL file and then written to the database file, called checkpointing writes. On the other hand, NAND flash-based SSDs, which have an inherent indirection software layer, called flash translation layer (FTL), become commonplace in modern storage systems. Innovative SSD designs have been proposed to eliminate the WAL overheads by exploiting the FTL, such as providing an atomic write interface or utilizing its address remapping. However, these designs introduce significant performance overheads of maintaining and persisting extra transactional information to guarantee the transactional atomicity or mapping consistency.**

In this paper, we propose *single-write WAL (SW-WAL)*, a novel cross-layer design, to eliminate WAL-induced duplicate writes on SSDs with minimal overheads. The SSD exposes an address remapping interface to the host, through which the checkpointing writes can be completed without conducting real data writes. To ensure the transactional atomicity and mapping consistency, we make the SSD aware of the transactional writes to the WAL file. Specifically, when transactional data are written to the WAL file, both transactional and mapping semantics are delivered from the host to the SSD and persisted in relevant flash pages as housekeeping metadata without any extra overheads. We implement a prototype of SW-WAL, which runs a popular database SQLite on an emulated NVMe SSD. Experimental results show that SW-WAL improves the database performance by up to 62% compared with original SQLite that bears the WAL overheads and up to 32% compared with the state-of-the-art design that eliminates the WAL overheads.

I. Introduction

Many databases adopt *write-ahead logging (WAL)* to provide transactional atomicity and thus guarantee data consistency despite sudden system crashes. Instead of updating a database file directly, the WAL appends data updates of a transaction in a separate log, called redo log or WAL file. After the transaction is committed and the WAL file is flushed to persistent storage, the updates are applied to the database file through *checkpoint writes*. However, the WAL introduces duplicate writes, causing performance degradation [1].

On the other hand, flash-based *solid state drives (SSDs)* are replacing mechanical hard disks and become commonplace in modern storage systems. SSDs write/program data in units of flash pages, and a page cannot be rewritten unless it is erased. An erase operation is performed in units of flash blocks. Each block contains hundreds of flash pages and has limited write endurance. SSDs employ a *flash translation layer (FTL)* to perform out-of-place updates and maintain a *logical-to-physical (L2P)* page mapping table to translate host *logical page numbers (LPNs)* to *physical page numbers (PPNs)* on flash memory. The L2P mapping table is persisted on flash memory and also cached in the built-in DRAM of SSDs to accelerate the lookups [2]. When logical data pages are written to flash pages, the relevant LPNs are stored in the *out-of-band (OOB)* area of the flash pages, which is transparent to users. Such *PPN-to-LPN (P2L)* reverse mappings are necessary to maintain data consistency as detailed in Section II-B1. Furthermore, the FTL performs *garbage collection (GC)* when the free pages are consumed to a low watermark by writes. GC operations reclaim flash pages storing invalid data by migrating valid data pages in victim blocks and then erasing the blocks. GC overheads are a critical factor that degrades the performance and lifetime of SSDs [3], [4].

To eliminate duplicate writes caused by journaling mechanisms including WAL, two kinds of innovative SSD designs have been proposed by utilizing the FTL. First, an atomic write interface can be exposed by an SSD. Applications can update data through the atomic write interface, so journaling mechanisms can be entirely removed. However, these approaches either contradict the steal policy of buffer management which is used by default for high performance [5], [6] or introduce extra overheads of maintaining and persisting an additional transactional page mapping table [7] to guarantee the transactional atomicity.

Second, duplicate writes can be conducted without real data transfers and flash writes by directly modifying the L2P mapping table, called address remapping [8]–[12]. However, LPNs stored in the OOB area of remapped flash pages can not be modified due to the non-overwrite characteristic of flash memory. Thus, the P2L mappings become inconsistent with the L2P mapping table, which finally causes data losses. Note that it is not feasible to maintain the relevant P2L mappings of remapped flash pages in a separate and persistent log mainly due to high lookup overheads, as discussed in Section II-B1.

In this paper, we propose *single-write WAL (SW-WAL)*, a novel cross-layer design, to eliminate WAL-induced duplicate writes on SSDs with minimal overheads. The SSD exposes an address remapping interface to the host, through which the checkpointing writes can be completed without conducting real data writes. To ensure the transactional atomicity and mapping consistency, we make the SSD aware of the transactional

978-3-9819263-5-4/DATE21/©2021 EDAA

writes to the WAL file. Specifically, when transactional data are written to the WAL file, both transactional and mapping semantics are delivered from the host to the SSD and persisted in relevant flash pages as house-keeping metadata without any extra overheads. We perform a case study on the WAL mode of SQLite running on an emulated NVMe SSD to validate our proposed design. Experimental results show that our proposed design improves the bandwidth by up to 62% and reduces the writes to the flash by up to 45% compared with original SQLite that suffers from the WAL overheads, and improves the bandwidth by up to 32% and reduces the writes to the flash by up to 23% compared with a state-of-the-art design that eliminates the WAL overheads. Our design can also be easily implemented and applied to other databases that adopt WAL and in file system journaling such as in EXT4.

The remainder of the paper is organized as follows: Section II-B introduce the background and motivation of the research; Section III describes the design of our proposed design and Section IV presents a case study based on SQLite and FEMU; Section V validates and evaluates the design; Section VI introduces related work; Section VII concludes our work.

II. Background and Motivation

A. Write-ahead logging

We take SQLite as an example to illustrate the WAL mechanism. SQLite is a lightweight database which is widely used in embedded systems such as smart phones. WAL is utilized in SQLite to provide transactional atomicity and thus guarantee data consistency.

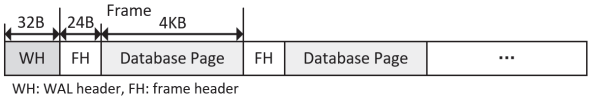


Fig. 1. The WAL file format of SQLite.

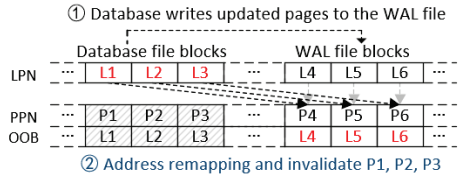
When a transaction updates data pages (P1...Pn) of the database file, the changes (P1_new...Pn_new) are appended to the WAL file instead of directly applied to home locations in the database file. When the WAL file is full or the database file is closed, committed changes in the WAL file are rewritten to their home locations in the database file (i.e., checkpointing writes). Figure 1 shows the default structure of the WAL file of SQLite. The WAL file consists of a 32 bytes WAL header and multiple frames. A frame contains a 24 bytes frame header and a database page. The frame header records the page number of the database page, a commit flag and some checksum data.

B. Motivation

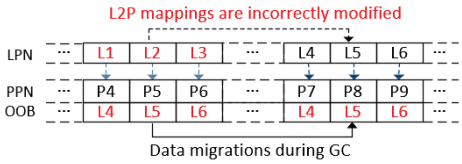
In this subsection, we analyze the problems of conventional SSD designs providing an atomic write interface or utilizing address remapping.

1) Mapping Inconsistency Caused by Address Remapping: A flash page has a data area storing user data and an OOB area storing house-keeping metadata including the LPN. In the common case, such a P2L mapping stored in the OOB area is consistent with the LPN of this page in the L2P mapping table.

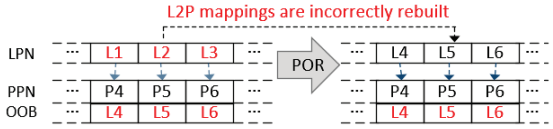
Maintaining the P2L mapping is necessary for two reasons. Firstly, when a physical page is relocated to another place (e.g. in garbage collection), we must look up the P2L mapping to identify the logical page mapped to this physical page and modify its L2P mapping to point to the new location. Secondly, the FTL needs to rebuild the latest L2P mappings based on the persistent P2L mappings after a sudden system crashes or power outage, where the latest L2P mappings cached in the DRAM may have gotten lost [13].



(a) Conventional address remapping.



(b) Incorrect L2P mappings after GC.



(c) Incorrect L2P mappings after the power-off recovery.

Fig. 2. Conventional address remapping and the mapping inconsistency issue.

Address remapping has been utilized in many prior studies to reduce duplicate writes on SSDs [8]–[12], [14]–[16], including checkpointing writes in the WAL. As illustrated in Figure 2, updated pages (L1', L2', L3') are written to the WAL file. Then address remapping modifies the L2P mapping table and remaps the home-location LPNs (original LPNs in the database file), i.e. (L1, L2, L3), to PPNs of updated data in the WAL file, i.e. (P4, P5, P6). However, the LPNs stored in the OOB area of remapped pages, i.e. (L4, L5, L6) marked in red, become inconsistent with the LPNs of these page in the L2P mapping table, i.e. (L1, L2, L3) marked in red. These stale P2L mappings will result in incorrect L2P mappings and thus data losses in the SSD during internal data migrations or the *power-off recovery (POR)*, as shown in Figure 2(b) and Figure 2(c).

A straightforward solution is to maintain and persist a remapping log by appending the latest P2L mappings, which has been adopted in some prior works [10], [14], [16]. However, such a method is ill-considered for three reasons. First, maintaining a remapping log in the volatile DRAM still suffers from the risk of data losses after a sudden power outage. Second, persisting the remapping log to the flash causes extra writes and performance losses. Furthermore, as address remapping operations are conducted over time, the size of the remapping log continuously increases, which leads to increased and finally prohibitively high lookup overheads (i.e. scanning the entire remapping log) during data migrations of GC. Suppose that the data page size is 4KB and each remapping entry takes 12 bytes [14], [16]. when 10GB and 100GB data page have been remapped, the remapping log size is at least as large as 30MB

and 300MB, respectively. WAL-SSD [12] stores home-location LPNs in the OOB area of remapped flash pages to guarantee mapping consistency. However, it uses individual commands to send home-location LPNs to the SSD, which must be completed before writing the relevant pages to the flash. This incurs extra performance overheads. What's more, it doesn't guarantee the transactional atomicity during SSD POR, which is discussed in Section III-C.

2) Problems in Atomic-write SSDs: Prior works have presented innovative SSDs that expose a transactional atomic write interface to the host by leveraging the inherent indirection layer inside SSDs. In these approaches, the SSD is not aware of multiple write requests that belong to the same transaction. Thus, they only guarantee limited transactional atomicity either on a per-call basis or when the whole set of pages are flushed from the buffer at once [5], [6]. This requirement contradicts the steal policy of buffer management adopted by default in most database systems for performance reasons.

Another work, X-FTL [7], adds an additional transactional logical-to-physical page mapping table (or X-L2P table in short) in the SSD to guarantee the transactional atomicity without compromising the steal policy. The X-L2P table records L2P mappings of pages of uncommitted transactions and the original L2P mappings of these pages are kept unchanged until the relevant transaction is committed. However, X-FTL needs to persist the X-L2P table to the flash memory whenever a transaction is committed, which not only increases writes to the flash but also degrades the performance. Especially, studies [7], [17] show that the write sizes of most transactions in SQLite are smaller than 20KB, implying significant persistence overheads of the X-L2P table. We conducted experiments to show extra writes caused by the X-L2P table in X-FTL. Detailed configurations are presented in Section V-A. The experiments include inserting a stream of key-value entries into a SQLite database file and then updates these entries, and the size of each entry is about 20KB. Experimental results show that persisting the X-L2P table increases writes to the flash memory by 11%∼23% (the relevant figure is not shown) and subsequently incur more GC operations.

In summary, these analyses motivate us to propose a novel design whose goal is to utilize the SSD address remapping to eliminate the duplicate WAL writes and maintain the mapping consistency and transactional atomicity with minimal performance overheads.

III. Design

A. Overview

In this paper, we propose SW-WAL, a novel cross-layer design. The SSD exposes an address remapping interface to the host, through which the checkpointing writes can be completed without conducting real data writes. We add a new command to the interface command set:

- **remap (src_lpn, dst_lpn, N)**: it modifies the L2P mapping table of the SSD by remapping N contiguous LPNs starting from *dst_lpn* to PPNs previously mapped to N contiguous source LPNs starting from *src_lpn* and leaving the source LPNs unmapped.

Home-location LPNs (e.g., (h-L1, h-L2, h-L3) in Figure 3) can be remapped to PPNs of transactional data pages written to the WAL file. The original LPNs (e.g., (L1, L2, L3) in Figure 3) of transactional data pages will be left unmapped. Adding a customized command in the command set is supported by most interface techniques, such as NVMe, an advanced interface technique designed for SSDs and has become mainstream in various storage systems.

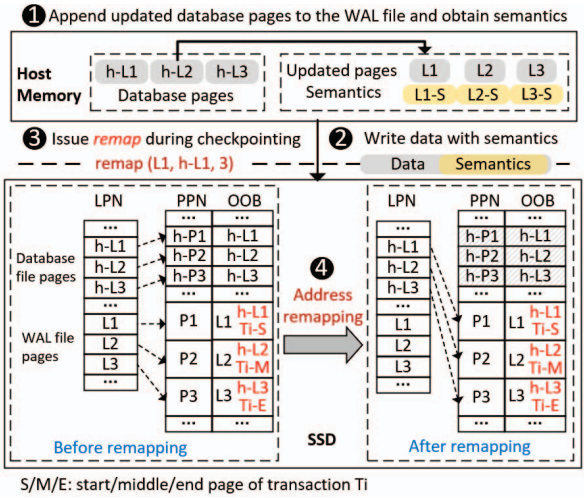


Fig. 3. Overview of SW-WAL.

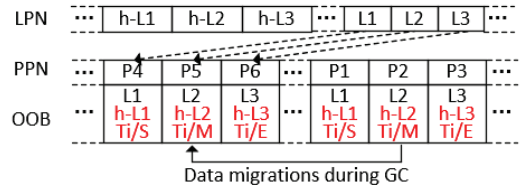
To ensure the transactional atomicity and mapping consistency, SW-WAL makes the SSD aware of the transactional writes to the WAL file. Specifically, we observe that checkpointing writes have deterministic destination addresses, i.e. home locations in the database file. When transactional data are written to the WAL file, both transactional and mapping semantics are delivered from the host to the SSD and persisted in the OOB area of relevant flash pages as house-keeping metadata.

As shown in Figure 3, when a transaction writes data pages to the WAL file, whose LPNs are (L1, L2, L3), SW-WAL obtains necessary mapping and transactional semantics (L1-S, L2-S, L3-S) (hereinafter referred to as MT-semantics) from the file system and the database, including their home-location LPNs (h-LPNs) in the database file, transaction IDs (TxIDs) and page identifiers. The page identifier (2 bytes) contains an offset of a page in the transaction and a 1-bit end flag indicating whether the page is the end page of the transaction. For other non-transactional data without h-LPNs, their MT-semantics are filled with *NULL* value and won't be remapped by SW-WAL. Usually, the data page size is 4KB and the ratio of OOB area size to data area size is about 1/8 in the latest flash chips [18]. For example, if the flash page contains a 16KB data area and a 2KB OOB area, 4 data pages can be stored in the data area and the OOB area usually has sufficient space to store the MT-semantics, whose size is 40 bytes (4×(4 bytes h-LPN + 4 bytes TxID + 2 bytes page identifier)).

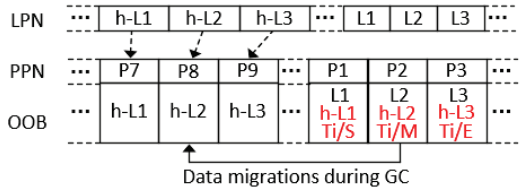
B. Maintaining the Mapping Consistency in GC

Thanks to the mapping semantics persisted in the OOB area of flash pages, SW-WAL avoids the mapping inconsistency

between L2P mappings and P2L mappings caused by address remapping. When valid pages of flash blocks chosen by GC are migrated to new flash pages, FTL should scan the OOB area of migrated pages to obtain correct reverse mappings. For pages without MT-semantics in the OOB area, the FTL can directly acquire the correct LPNs. For pages with MT-semantics in the OOB area, the FTL looks up the corresponding L2P mappings to identify whether these pages have been remapped or not. If these pages have not been remapped, the original LPNs and MT-semantics in the OOB area should be written to the OOB area of new flash pages. For example, as shown in Figure 4(a), transactional data pages (P1, P2, P3) have not been address remapped and are migrated to new flash pages (P4, P5, P6) during GC. Then the MT-semantics are written to the OOB area of new pages and the FTL modifies the L2P mapping table, mapping (L1, L2, L3) to new flash pages (P4, P5, P6). Otherwise, if these pages are remapped to their home-location LPNs, then only home-location LPNs will be written to the OOB area of new flash pages and the corresponding L2P mappings are modified. As illustrated in Figure 4(b), pages (P1, P2, P3) are remapped to their home-location LPNs (h-L1, h-L2, h-L3), then only (h-L1, h-L2, h-L3) will be written to the OOB area of new flash pages (P7, P8, P9) when (P1, P2, P3) are migrated.



(a) GC before address remapping.



(b) GC after address remapping.

Fig. 4. Maintaining the mapping consistency in GC.

C. Power-Off Recovery

Upon a sudden system crash or power outage, SW-WAL ensures both the transactional atomicity and mapping consistency by scanning the OOB area of all flash pages. For pages without MT-semantics in the OOB area, the FTL directly maps LPNs in the OOB area to the PPNs of these pages, such as P1 in Figure 5. SW-WAL identifies whether all pages of a transaction have been persisted to the flash or not through page identifiers in the OOB area of flash pages. If all pages of a transaction have been persisted to the flash, such a complete transaction can be applied to the database file. Therefore, the FTL maps h-LPNs in the OOB area to the PPNs of these pages, such as the transaction Ti consisting of pages (P4, P5, P6) in Figure 5. On the contrary, incomplete transactions can not be applied to the database file to guarantee the transactional atomicity. Then the FTL maps original LPNs (WAL file LPNs) in the OOB area to the PPNs of these pages, such as transaction Tj with pages (P10, P11) in Figure 5.

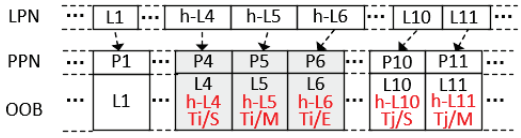


Fig. 5. Power-off recovery of SW-WAL.

IV. Case Study: SQLite on an NVMe SSD

To demonstrate the effectiveness of our design, we implement SW-WAL on an NVMe SSD hosting EXT4 and SQLite (version 3.31.0). NVMe supports a host metadata feature, which allows self-defined metadata to be transferred from the host to the SSD along with written LBA data. In this paper, we directly utilize NVMe host metadata support to deliver a few bytes of semantics for each transactional data page between the host software and the SSD.

A. Changes Made in SQLite

One challenge is that data pages appended to the WAL file are not aligned with the logical blocks of the file system (as illustrated in Figure 1) and the address remapping can not be successfully performed. Therefore, we modify the WAL file format, i.e., place the WAL header and frame headers in the front of the WAL file and append database pages from a 4KB-aligned offset. Since SQLite performs checkpointing when committed transactions in the WAL file accumulate more than 1000 frames, 10 blocks are reserved for relevant headers in the front of the WAL file, as illustrated in Figure 6.

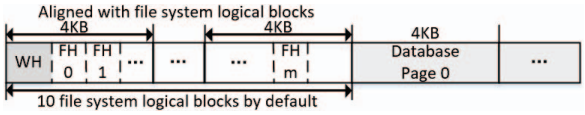
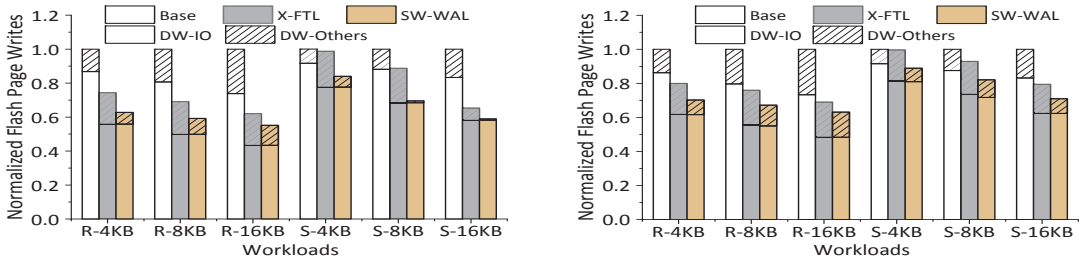


Fig. 6. Modified SQLite WAL file format.

We extend the *ioctl* system call to replace the *OsWrite* function in SQLite to write a database page to the WAL file and pass necessary information from SQLite to the kernel. For inserted data pages that don't have home-location LPNs, we fill their MT-semantics as *NULL* value. We also extend the *ioctl* system call to issue the *remap* command to perform address remapping, replacing the procedure of reading the WAL file and writing the database file during the checkpointing of SQLite.

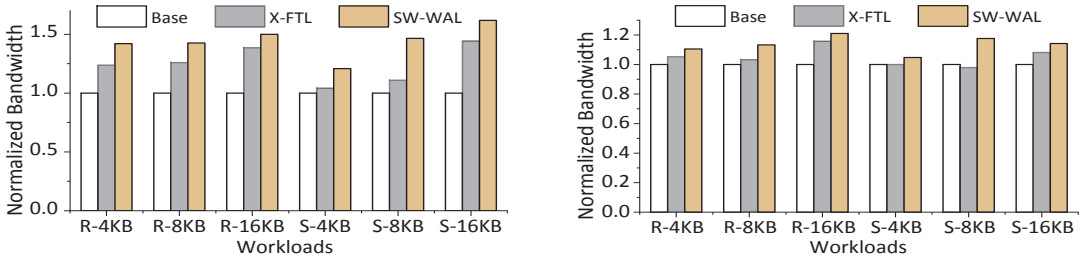
B. Changes Made in the Host System Kernel and FTL

Since SQLite doesn't generate a unique ID for each transaction, we deliver other semantics to the SSD and allow the FTL to generate a unique ID for each transaction. When SQLite writes a frame, including a header and a database page, to the WAL file, SW-WAL passes the file descriptor (*fd*) of the database file, page number (*pgno*) of the database page in the database file, page offset (*pgoff*) of the database page in the WAL file and an end flag (*pgfg*) to the kernel space through the extended *ioctl*. Finally, these semantics are transferred to the SSD by the NVMe host metadata interface. After receiving the (*ino, pgoff, pgfg and h-LPN*) in the SSD, the FTL generates the



(a) synchronous=NORMAL (b) synchronous=FULL

Fig. 7. Flash page writes analysis. (*R/S-mKB* of different workloads means *random/sequential* updates and the value size is *m* KB. *DW-IO* means data writes fron host IO and *DW-Others* includes data writes from GC operations and persisting the X-L2P table in X-FTL).



(a) synchronous=NORMAL (b) synchronous=FULL

Fig. 8. SQLite Performance analysis.

page identifiers and a unique *TxID* for each transaction. Then the *TxID*, *page identifier* and *h-LPN* are stored in the OOB area of each transactional page. Implementing SW-WAL prototype modifies only fewer than 300 lines of code (LoC) in the SQLite and host system kernel.

V. Evaluation

A. Experimental Setup

We use FEMU, a widely used QEMU-based emulator [19], to obtain an NVMe SSD and add support for host metadata delivery and the *remap* command. The kernel of the FEMU host system is Linux 4.15 and the file system of the mounted SSD is EXT4. In the experiments, a 32GB SSD with 4GB over-provisioned space is emulated. The SSD has two channels and four dies in each channel. The flash page size is 4KB and latencies of page read, page write and block erase are 50µs, 500µs and 5ms [20], respectively.

The selected benchmark [21] is released by Google and used to test performance of SQLite. In our experiments, we let the selected benchmark for SQLite (*db_bench_sqlite3.cc*) issue a stream of entries, each for writing a key and its associated certain sized value into the database. To obtain an aged SSD and stable performance results, we first insert a stream of entries into the database file and let GC operations triggered. Then we randomly/sequentially update entries and collect the statistic data during the update stage to get the experimental results.

We tested the influence of transaction sizes, e.g. 4KB value, 8KB value and 16KB value. We also conducted experiments under different synchronous settings of SQLite, i.e. NORMAL and FULL [22]. In the NORMAL mode, the WAL file is synchronized before each checkpoint and the database file is synchronized after each completed checkpoint. This setting is a good choice for most applications running in WAL mode. In the FULL mode, SQLite uses the *xSync* function of the VFS after each transaction commit. We compare SW-WAL with both original SQLite, which bears WAL overheads and is referred to as Base, and X-FTL [7], which removes the WAL overheads but needs to maintain and persist an additional transactional page mapping table for transactional atomicity.

B. Experimental Results

Figure 7 illustrates the normalized flash page writes, which is a critical factor that limits the performance and lifetime of SSDs. Figure 8 shows the SQLite performance of three schemes.

Flash Page Writes: Compared with Base, X-FTL and SW-WAL has less host IO data writes (*DW-IO*) owing to reduced duplicated checkpointing writes. However, X-FTL needs to persists the X-L2P table to the flash whenever receiving a page with the end flag, which increases writes to the flash and subsequently incurs more GC operations. Especially in the sequential update workloads when the size of transactions is small (value size is 4KB or 8KB), X-FTL has more additional flash page writes (*DW-Others*) than Base and SW-WAL. When the synchronous setting is NORMAL, SW-WAL reduces 16%∼45% flash page writes compared with Base, and reduces 11%∼23% flash page writes compared with X-FTL, as shown in Figure 7(a). When the synchronous setting is FULL, the *xSync* is called to persist data to the flash whenever a transaction is committed, which leads to numerous writes to the flash and extremely low performance. SW-WAL reduces 11%∼37% flash page writes compared with Base, and reduces 9%∼14% flash page writes compared with X-FTL, as shown in Figure 7(b). Note that SW-WAL has higher performance gain and less flash page writes compared with X-FTL when the value size is small. More transactions will be recorded in the WAL file if the value

size is small, which causes more additional X-L2P table writes when the WAL file is synchronized.

SQLite Performance: When the synchronous setting is NORMAL, SW-WAL improves the performance by 20%∼62% compared with Base, and improves the performance by 8%∼32% compared with X-FTL, as shown in Figure 8(a). When the synchronous setting is FULL, SW-WAL improves the performance by 5%∼21% compared with Base, and improves the performance by 5%∼25% compared with X-FTL, as illustrated in Figure 8(b). In the FULL mode and S-8KB workload, X-FTL even has worse performance than Base as it has more extra flash writes than Base. The performance overheads caused by the metadata delivery are also evaluated by using *fio* to write data and deliver metadata to the SSD. Experimental results show that the host metadata delivery causes negligible performance degradation (up to 0.5%) when the metadata size is 12 bytes (in our experiments, the size of delivered semantics of a 4KB data page is 12 bytes). The relevant figure is not shown due to the limit of space.

VI. Related Work

Address remapping has been utilized in many prior studies to reduce duplicate writes on SSDs. Some approaches modify the mapping table to remap home-location LPNs of data in the database file to the PPNs of pages storing updated data in the WAL file, eliminating duplicate checkpointing writes [8], [9], [11], [16]. Ni et al. map new LPNs to the PPNs of pages storing original data, replacing data copy operations in the rollback journaling [10]. Hahn et al. utilize address remapping to achieve zero-copy logical defragmentations [14]. K. Han et al. leverage address remapping to eliminate duplicate writes in WAL and stores home-location LPNs in the OOB area of remapped flash pages [12].

Prior works have presented innovative SSDs that expose an atomic write interface to the host. Some approaches, such as transactional flash [6], store additional transactional information in the OOB area of flash pages to guarantee transactional atomicity. However, these approaches support the atomicity of a multi-page transaction only on a per-call basis or flushing all data pages of the transaction from the buffer at once [5], [6]. By contrast, SW-WAL supports general transactional atomicity, where the data pages in a transaction can be written to the flash at any time, by bridging the gap on transactional semantics between the host and SSD. Another work, X-FTL [7] exposes an atomic write interface to the host and maintains an additional transactional page mapping table in the FTL to provide transactional atomicity without compromising the steal policy. Drawbacks of these approaches have been discussed in Section II-B.

VII. Conclusion

In this paper, we propose SW-WAL, a novel cross-layer design, to eliminate the WAL-induced duplicate writes on SSDs with minimal overheads. The SSD exposes an address remapping interface to the host to complete the checkpointing without conducting real data writes. To ensure the transactional atomicity and mapping consistency, SW-WAL delivers mapping and transactional semantics from the host to the SSD and persists these semantics in relevant flash pages as housekeeping metadata without any extra overheads. We implement an prototype of SW-WAL in SQLite running on an emulated NVMe SSD of FEMU. The experimental results show that our proposed design can reduce writes to the flash and improve the system performance, compared with the original SQLite and an state-of-the-art design that eliminates the WAL overheads.

Acknowledgment

This work was supported in part by the NSFC under Grant No. 61902137, No. 61821003 , No. 61872413, No. U1709220, in part by Key_Area Research and Development Program of Guangdong Province No. 2019B010107001, in part by National Key Research and Development Program of China No.2018YFB1003305, No.2018YFA0701800, in part by the 111 Project (No. B07038), in part by the Project funded by China Postdoctoral Science Foundation.

References

[1] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *FAST*, 2014.
[2] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS*, 2009.
[3] C. Min, K. Kim, H. Cho, and et al., "SFS: random write considered harmful in solid state drives," in *FAST*, 2012.
[4] H. Kim, D. Shin, Y. Jeong, and et al., "SHRD: improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device," in *FAST*, 2017.
[5] P. Sunhwa, Y. Ji, and O. Seong, "Atomic write ftl for robust flash file system," in *ISCE*, 2005.
[6] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *OSDI*, 2008.
[7] W. Kang, S. Lee, B. Moon, and et al., "X-FTL: transactional FTL for sqlite databases," in *SIGMOD*, 2013.
[8] H. Choi, S. Lim, and K. H. Park, "JFTL: A flash translation layer based on a journal remapping for flash memory," *TOS*, vol. 4, no. 4, pp. 14:1–14:22, 2009.
[9] Z. Weiss, S. Subramanian, S. Sundararaman, and et al., "Anvil: Advanced virtualization for modern non-volatile memory devices," in *FAST*, 2015.
[10] F. Ni, X. Wu, W. Li, and et al., "Leveraging ssd's flexible address mapping to accelerate data copy operations," in *HPCC/SmartCity/DSS*, 2019.
[11] D. Kang, G. Oh, D. Kim, and et al., "When address remapping techniques meet consistency guarantee mechanisms," in *HotStorage*, 2018.
[12] K. Han, H. Kim, and D. Shin, "WAL-SSD: address remapping-based write-ahead-logging solid-state disks," *IEEE Trans. Computers*, vol. 69, no. 2, pp. 260–273, 2020.
[13] D. Ma, J. Feng, and G. Li, "Lazyftl: a page-level flash translation layer optimized for NAND flash memory," in *SIGMOD*, 2011.
[14] S. S. Hahn, S. Lee, C. Ji, and et al., "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *ATC*, 2017.
[15] Y. Jin, H. Tseng, and et al., "Improving SSD lifetime with byte-addressable metadata," in *MEMSYS*, 2017.
[16] G. Oh, C. Seo, R. Mayuram, and et al., "SHARE interface in flash storage for relational and nosql databases," in *SIGMOD*, 2016.
[17] D. Q. Tuan, S. Cheon, and Y. Won, "On the IO characteristics of the sqlite transactions," in *MOBILESoft*, 2016.
[18] Y. Zhou, F. Wu, and Z. L. et al., "SCORE: A novel scheme to efficiently cache overlong eccs in NAND flash memory," *TACO*, vol. 15, no. 4, pp. 60:1–60:25, 2019.
[19] H. Li, M. Hao, M. H. Tong, and et al., "The CASE of FEMU: cheap, accurate, scalable and extensible flash emulator," in *FAST*, 2018.
[20] B. S. Kim, J. Choi, and S. L. Min, "Design tradeoffs for SSD reliability," in *FAST*, 2019.
[21] Database microbenchmarks. [Online]. Available: http://www.lmdb.tech/bench/microbench/
[22] Synchronous mode in SQLite. [Online]. Available: https://www.sqlite.org/pragma.html