Removing Duplicated Writes at DB Checkpointing with File System-Level Block Remapping

Daejun Park
College of Information and Communication Engineering,
Sungkyunkwan University
Suwon, Korea
Tel: +82-31-299-4662
pdaejun@skku.edu

Dongkun Shin
College of Information and Communication Engineering,
Sungkyunkwan University
Suwon, Korea
Tel: +82-31-299-4662
dongkun@skku.edu

ABSTRACT

Database systems use the journaling techniques in order to guarantee database consistency even at system crashes. Since the journaling needs duplicated write operations, it exhausts the lifetime of NAND flash-based storage device. In this paper, we propose a novel file system-level block remapping technique, which changes only the file system metadata at database checkpointing without any data write operations. Experiments show that the proposed scheme can reduce the write traffic on storage device by 17% on average.

Keywords

database journaling; file system; NAND flash memory

1. INTRODUCTION

NAND flash memory-based storage systems have the advantages of low power consumption, non-volatility, fast random access performance, and durability. However, flash memory chip has a limited number of program/erase (P/E) cycles. If a flash memory block is programmed and erased repeatedly as the number of maximum P/E cycles, the block cannot be further utilized. Recently, the semiconductor process technology is scaled down and multi-level cell flash memory devices are widely used. As a result, the storage capacity has been increased significantly by sacrificing the maximum P/E cycles of flash memory. Recent mobile devices such as smartphones generally use the SQLite DBMS in order to manage various user data and system configurations easily. The database files are stored at NAND flash memory-based storages such as eMMC and SD card. Therefore, the storage performance can affect the performance of mobile devices. SQLite supports two journaling modes: rollback journal and write-ahead log (WAL) [1]. The rollback journal scheme first copies the original pages of DB file to the rollback journal file before updating the DB pages. If a system crash occurs during the update operation, the journaling scheme can restore the old data with the rollback journal. The WAL scheme appends a new data to a WAL file without modifying the original DB file. When the WAL file is filled with many logs, SQLite copies all the valid log pages of the WAL file to the original DB file, which is called checkpointing. These journaling schemes ensure the database consistency with duplicated write operations on the original DB file and the journal file. Such duplicated write operations shorten the lifetimes of NAND flash storage devices. Considering that the data to be written at checkpointing have already been written in the journal file under the DB journaling schemes, we can remove the duplicated write operations at checkpointing by changing only the block mapping information of file system. Then, we can mitigate the write traffic on the storage device. File system maintains a block mapping table which can translate a file offset into the storage block address. By modifying the mapping table, we can remap the journal file blocks to the original DB file blocks without explicit file system read and write operations. In this paper, we propose a file system-level block remapping technique for DB checkpointing, targeting for EXT4 file system.

CF'15, May 18-21, 2015, Ischia, Italy
ACM 978-1-4503-3358-0/15/05.
http://dx.doi.org/10.1145/2742854.2742891

2. BLOCK REMAPPING FOR CHECKPOINTING

EXT4 file system manages an inode for each file, which has the attributes of each file and the block mapping table for the file. By referring to the block mapping table, the file system can access the corresponding storage block for a given file offset. The proposed block remapping technique changes the block mapping tables of DB file and journal file at checkpointing. For the purpose, we added the *remap* system call at EXT4 file system, which can exchange the blocks of two different files. Figure 1 shows the file system metadata changes in the block remapping technique.

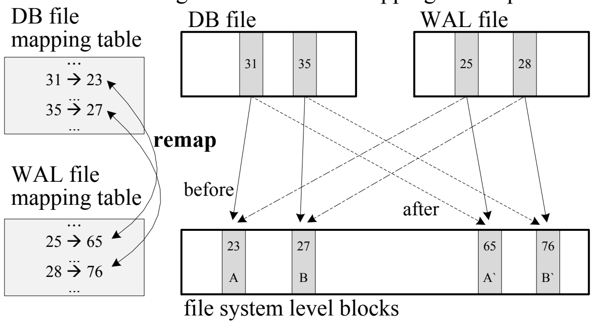
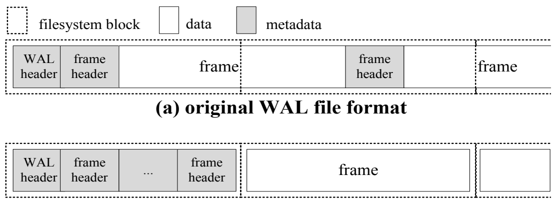


Figure 1: block remapping operations.

The original DB file has the data of A and B at the file offsets of 31 and 35, respectively. By DB update operations, the WAL file has the data of A′ and B′, which are modified data of A and B, respectively. When the checkpointing is issued, the original SQLite reads A′ and B′ from the WAL file, and writes them at the DB file. However, our modified SQLite sends the remap system call with the file offsets to be exchanged as arguments. For the remap system call, EXT4 file system exchanges the blocks of DB file and WAL file by modifying the mapping tables. The block mapped to the file offset 31 of the DB file is changed from the block address 23 to the block address 65 by the block remapping operation in Figure 1. Then, the DB file can access the new data after the block remapping.

For the block remapping, the DB pages should be aligned with the block size of file system. Generally, Linux uses 4KB of block size, and the data page size of SQLite DB can be configured as 4KB. However, as shown in Figure 2(a), the WAL header and frame header of WAL file are smaller than 4KB, and thus the data pages are miss-aligned at 4KB boundary. To solve this problem, we modified the file format of WAL as shown in Figure 2(b). By gathering multiple frame headers at the first 4KB-sized block, the data pages can be located at the 4KB-aligned addresses.



(a) original WAL file format

(b) modified WAL file format

Figure 2: WAL file formats.

When SQLite creates a WAL file, EXT4 file system generally allocates contiguous storage blocks for the file. During the DB modifications, SQLite writes data at the WAL file sequentially. After the checkpointing, SQLite reuses the WAL file, and the file is overwritten sequentially. Therefore, the write pattern on the WAL file is always sequentially. However, under the block remapping technique, since the WAL file exchanges its data blocks with the DB file, the blocks of WAL file will be fragmented after several checkpointing operations. The fragmented WAL file will provide a low write performance since flash memory devices have a low random write performance. To solve the problem, our scheme reserves a large sequential storage space for a WAL file, and uses an address translation layer between the WAL file and EXT4 file system as shown in Figure 3. The translation layer changes the start block address of WAL file after each checkpointing within the sequentially reserved space by a user-transparent fashion. The remapped start block is the next block to the last written block at the previous checkpointing. Therefore, the written data of WAL file are appended sequentially at the reserved space even after the checkpointing. With the translation layer, we can implement an append-only WAL file without modifying the original SQLite. The translation layer is implemented with a stackable file system, wrapfs [2]. By using the translation layer instead of reallocating sequential blocks for WAL file, we can remove the metadata handling overhead for block allocation since the translation layer pre-allocates the data blocks for WAL file when it is created. Even when there is a system crash during the checkpointing, the original SQLite can redo the checkpointing since the WAL file still remains without any modifications. However, the block remapping technique modifies the blocks of WAL file. Therefore, the blocks of DB file and WAL file may be exchanged partially at the sudden failure. We can solve the problem by exploiting the file system journaling technique which can guarantee an atomic change on file system metadata. That is, all the changes on block mapping tables can be written atomically at the storage. Therefore, the database consistency can be guaranteed under the block remapping scheme.

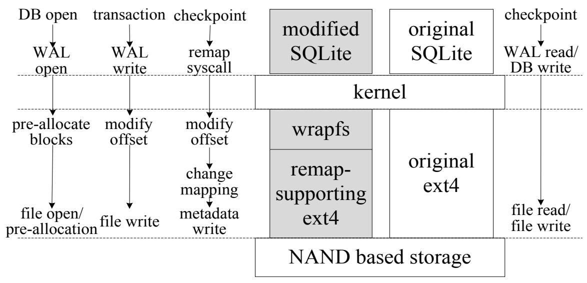


Figure 3: Architecture of block remapping scheme.

3. EXPERIMENTS

We evaluated the proposed technique at an Android-based smartphone device equipped with 32-GB eMMC storage. With the Mobibench SQLite benchmark [3], we observed the changes on the storage write traffic and the DB performance by the block remapping technique. 8KB records are inserted at DB files during the experiments. We implemented two versions of block remapping technique, DB remap and DB remap+. The first uses only the block remapping technique without the pre-allocation technique. The second uses the append-only pre-allocated WAL additionally. The size of pre-allocated WAL file is 150 MB.

Since the DB remap scheme changes only the file system metadata without extra DB write operations, it reduces the amount of written data by 17% compared with the original WAL scheme as shown in Figure 4. However, the performance of DB remap is decreased by 5% compared with the original WAL scheme. This is because the WAL file is fragmented by the block remapping. However, the DB remap+ scheme improved the performance by 11% by removing the fragmentation problem. However, the pre-allocation of WAL file can be a system overhead. We will optimize the pre-allocation technique as a future work.

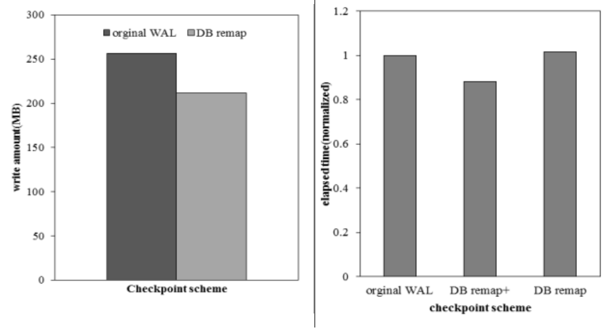


Figure 4: Write amount and elapsed time.

4. ACKNOWLEDGEMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2013R1A1A2A10013598).

5. REFERENCES

[1] http://www.sqlite.org/wal.html.

[2] E. Zadok, I. Badulescu, and A. Shender, "Extending File Systems Using Stackable Templates", *USENIX Annual Technical Conference, General Track'99*, pp.57-70, 1999.

[3] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "AndroStep: Android Storage Performance Analysis Tool", Software Engineering (Workshops), pp.327-340, 2013