Stackable Transactional File System Using Kernel-Level WAL

DAEJUN PARK\textsuperscript{1} AND DONGKUN SHIN\textsuperscript{2}, (Member, IEEE)
\textsuperscript{1}Samsung Electronics, Suwon 16677, South Korea
\textsuperscript{2}Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
Corresponding author: Dongkun Shin (dongkun@skku.edu)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant NRF-2021R1A2C2095125.

ABSTRACT
User applications that require transactional file access generally use user-level transactional protocols such as logging, atomic rename, since the POSIX interface does not support transactional file operations. The user-level protocols are complex and require specific persistence properties of file systems for correctness. If the kernel provides transactional access to file data, application developers can avoid the complex and error-prone application-level protocols. Many previous studies on kernel-level transactional support have several critical limitations such as complex APIs, file system-dependence, and low transaction concurrency. In this paper, we propose a kernel-level write-ahead logging (WAL) technique, called KWAL. KWAL maintains user-transparent WAL files and redirects transactional IOs to them. Since KWAL is implemented as a stackable file system, it can be mounted over any native file systems, without modifying them. To provide a higher transaction concurrency while guaranteeing isolation, KWAL supports byte-level write conflict detection and repeatable read isolation. We also utilize a block remap technique to resolve the duplicate write problem of WAL-based transaction scheme. To demonstrate the usefulness of KWAL, we implemented a simple key-value store using KWAL APIs, fixed the wrong transaction implementation of the Git application, and simplified the implementation of the storage engines of MySQL and SQLite.

INDEX TERMS Transactional file system, write-ahead log, stackable file system, concurrency.

I. INTRODUCTION

The transactional interface to files is an important requirement to provide the integrity of application data. For example, the Gnome desktop environment uses many configuration files to store application state, and a consistent application state must be restored from these files even after sudden system crashes [1]. Despite the existence of database interfaces, many applications implement their own transactional file access protocols through the POSIX system call interface. However, the POSIX interface does not support transactional file system operations [2], and the exact specifications of how storage state is updated by a file system in the event of a crash are unclear [3], [4]. The existing file system’s journaling support is primarily concerned with protecting file system metadata consistency rather than application data integrity.

Therefore, several application-level protocols have been used, i.e., write-ahead logging (WAL), atomic rename, and rollback journaling, to implement a user-level transactional file interface. Unfortunately, the existing techniques for crash consistency result in complex protocols and require specific persistence properties of file systems for correctness such as atomicity and ordering. However, since each file system provides different persistence properties, many applications are crash-vulnerable [4]. For example, the atomic rename protocol creates a temporary file by copying the transaction target file, updates it, and renames the file to the original file. This protocol requires the ordering guarantee among the system calls of creat, write, and rename. Therefore, if developers do not understand the persistence properties of underlying file systems nor use the ordering and durability-related file system APIs such as fsync() appropriately, the crash consistency cannot be guaranteed. Instead of application-level protocols, applications may store their data through
well-designed database management systems (DBMS). However, the application data must follow database data structures, and the data can be accessed only through SQL or key-value query. A user-level transactional library such as Berkeley DB [5] and Stasis [6] also can be used. However, Berkeley DB provides complex user APIs and Stasis has the problems of redundant page cache and not supporting non-transactional processes.

If the kernel were to provide transactional access to file data, application developers would not have to use the complex and error-prone application-level protocols. There are many previous studies on kernel-level supports for transactional data access [1], [3], [7], [8], [9], [10], [11], [12], [13]. However, the existing techniques have the following limitations. First, the provided kernel-level APIs are complex and users must understand the internal logging mechanism for correct implementations, as shown in Valor [1], TxF [14], and QuickSilver [15].

Second, they are limited to specific file systems or depend on transactional storage devices. TxFS [10] leverages the journaling mechanism of the EXT4 file system. TxLFS [9], F2FS [8], and btrfs [11] utilize the out-of-place update policy of log-structured file systems or copy-on-write file systems to support the transactional file access. TxFlash [13], MARS [12], CFS [16], and TxSSD [17] rely on transaction supporting storage devices.

Third, transaction isolation is supported by sacrificing concurrency. Valor [1] uses a lock to prevent concurrent accesses to the pages or file of a transaction. Although TxFS [10] supports block-level write conflict detection and repeatable read isolation, a transaction can be aborted by false conflict owing to the block-level approach. For example, two transactions which access different data can conflict with each other if their data are in the same block. Since TxFS is implemented at the journaling system of EXT4, the write conflict check can be performed only at the block-level. Isotope [7] is a transaction system implemented at the block layer. Therefore, Isotope cannot detect false conflicts as with TxFS. To solve the problem, Isotope resorts to a special API, through which applications can mark byte range accessed by a transaction.

Considering the limitations of existing kernel-level solutions, we can summarize the requirements of kernel-level transactional systems as follows: First, the provided transactional APIs must be easy for developers who have no knowledge of in-kernel transaction implementations. Second, the transaction system must not rely on specific file systems or transactional storage devices. Third, transaction isolation must be supported without hurting concurrency and requiring user intervention. We propose a WAL-based kernel-level transactional file access technique, called KWAL, which satisfies all the aforementioned requirements. KWAL has the following unique features and advantages.

A. SIMPLE API
KWAL provides simple transactional APIs such as tx_start() and tx_commit(), and guarantees the atomicity and durability of file read/write operations between the two commands with the write-ahead logging technique. Although the functionality of KWAL is simple, it enables applications to avoid the complex user-level transaction protocols.

B. FILE SYSTEM-INDEPENDENT DESIGN
KWAL is implemented as a stackable file system [18] to integrate easily and seamlessly with existing file systems. Using the features of the stackable file system that can be mounted on another file system, KWAL has no restriction or dependency on file systems, and can be applied to any native file systems. Additionally, KWAL is a device-agnostic kernel-level technology, which does not require any modifications to legacy storage device.

To avoid handling the intrinsic persistence properties of a file system, KWAL supports only the transactional read and write operations for existing files. In other words, it does not guarantee the atomicity or ordering of file operations which incur block allocation and file system metadata modification. Such a limitation is insignificant, as the most serious crash vulnerabilities of applications are caused because the applications expect the atomicity of file updates [4]. Although several application-level transaction protocols based on logging, journaling or atomic rename invoke file creations and deletions, such protocols have been used against normal kernels. If the kernel supports transactional file access, it is not required to create and delete auxiliary files to update an existing file.

The internally managed WAL files, called KWAL files, are pre-allocated before transactions, and all the transaction handling metadata are encapsulated within the KWAL file. In addition, multiple transactions share a single KWAL file instead of having its own one. Therefore, no file system metadata changes occur during a transaction handling while minimizing the interaction with native file systems.

C. TRANSACTION ISOLATION WITH HIGH CONCURRENCY
KWAL provides repeatable read isolation and write conflict detection. Applications do not need to use a lock mechanism for isolation. Each in-flight transaction has its private version of data, which are changed to global data after transaction commit. Therefore, a transaction’s data are invisible to other transactions until they are committed, and multi-version concurrency is supported. Because KWAL is interposed between the virtual file system (VFS) and the native file system, it can identify the byte range of a transaction by observing write system calls. Therefore, a byte-level write conflict detection is possible without user intervention. Then, transaction concurrency can be improved by removing transaction aborts resulting from false conflicts.

D. HIGH PERFORMANCE
To optimize the transaction commit overhead, KWAL uses the no-force policy, i.e., the transaction target file is not updated immediately at transaction commit. Instead, the
In transactional systems, there is no need to modify the native file systems for transaction support. Each copy of file structure and inode structure of the native file system is maintained at the stackable file system. Figure 1 shows the overall architecture of the KWAL transaction system. KWAL manages multiple KWAL files transparent to the user and redirects any transaction-relevant IOs to the KWAL files to support isolation and atomicity of transactions. The redirection information from the original file block offset \(B_{org}\) to the KWAL file block offset \(B_{wal}\) is managed by the in-flight transaction table and committed transaction table.

### II. KWAL

#### A. OVERALL ARCHITECTURE AND API

Since KWAL is a stackable file system, all the transaction supports are implemented in the stackable file system and there is no need to modify the native file systems for transaction support. Each copy of file structure and inode structure of the native file system is maintained at the stackable file system. Figure 1 shows the overall architecture of the KWAL transaction system. KWAL manages multiple KWAL files transparent to the user and redirects any transaction-relevant IOs to the KWAL files to support isolation and atomicity of transactions. The redirection information from the original file block offset \(B_{org}\) to the KWAL file block offset \(B_{wal}\) is managed by the in-flight transaction table and committed transaction table.
is called, KWAL saves a value of their related metadata. Since each KWAL file is pre-allocated, the journal flush at journaling file systems can be avoided when the commit operation calls $\text{fdatasync}()$. The transaction logs are appended at the KWAL file to prevent data loss instead of using in-place updates. Whereas the data blocks are appended in the increasing order of block number starting from the first block, the metadata blocks are appended in the reverse order starting from the last block, as shown in Figure 1. The reason for separating the regions of data blocks and metadata blocks is to mitigate the fragmentation of the original transaction file after the remap-based checkpoint (§II-D).

The on-disk metadata of a KWAL file includes the block redirection entries (i.e., the array of $(B_{\text{org}}, B_{\text{wal}})$s). The committed flag, conflict flag, and time of the in-memory redirection entries are not recorded at the on-disk metadata. Instead, additional on-disk metadata are required for crash recovery: the magic number, the inode number of the target file, the number of redirected blocks, the byte-level target file size after transaction commit, the checksum values of each committed data block, the checksum value of overall metadata blocks ($CS_{\text{meta}}$), and the commit mark. These metadata can occupy multiple contiguous blocks. Since a native file system may not guarantee the atomicity of multi-block

VOLUME 10, 2022
110091

B. TRANSACTION-HANDLING METADATA
KWAL manages its own metadata for transaction handling. The in-flight transaction table and committed transaction are in-memory data structures for handling block redirection. KWAL uses block-level redirection since it is the basic IO unit of page cache and file systems. The in-memory transaction tables have the redirection information of each block modified by transactions.

Each redirection entry has the following fields. $B_{\text{org}}$ and $B_{\text{wal}}$ are the original file block number and the corresponding redirected KWAL file block number, respectively. They are used to redirect the transaction-relevant read and write requests to the KWAL files. The committed flag, which

D. Park, D. Shin: Stackable Transactional File System Using Kernel-Level WAL
write, KWAL has its own multi-block atomicity protocol. First, by checking \(CS_{meta}\) and the commit mark, multi-block atomicity of metadata blocks can be guaranteed. Second, the multi-block atomicity of data blocks can be checked by comparing data block values and their corresponding checksum values in the KWAL metadata. Therefore, the consistency and durability of a KWAL file can be guaranteed with only a single \(fdatasync()\) system call. Since the byte-level target file size is recorded on the on-disk KWAL metadata, KWAL can also guarantee the transaction consistency at append write operations.

KWAL does not support transactions for \textit{mmap}-ed data since the data changed by \textit{mmap()} are not transferred through the file system. One solution is to use the failure atomic \textit{msync()} proposed in [3]. However, \textit{mmap} data operations are implicit and the kernel cannot intervene, which increases the crash vulnerability of applications. Therefore, changing \textit{mmap()-style} I/Os to \textit{read()}/\textit{write()-style} I/Os can solve the vulnerability, as observed in Level DB [4].

C. TRANSACTIONAL OPERATIONS

Figure 2 shows an example of transactional file access using KWAL. Two user processes, A and B, access an existing file \textit{foo}. Process A notifies the kernel of the start of a transaction for the file through \textit{tx_start()}. Then, KWAL sets the \textit{on-going} flag of the file structure of \textit{foo}, and prepares the in-flight transaction table for the file. When process A writes the new data (D2) at the block offset 13 of the transaction target file \textit{foo} via \textit{write()}, KWAL redirects the write request to the KWAL file, and inserts the corresponding redirection entry to the in-flight transaction table. The original file \textit{foo} is not updated during the write operation.

When process A calls \textit{tx_commit()}, KWAL moves the redirection entries of the transaction to the committed transaction table, which can be shared by other processes. If duplicated original block numbers are found, the old entries are invalidated. The redirection entries are also written as metadata (M) at the KWAL file persistently. The \textit{on-going} flag of the file structure is cleared, and the \textit{need_CP} flag of the target inode is set to indicate that the checkpointing of the file is required. \textit{need_CP} is cleared after checkpointing.

Figure 3 shows the read and write handling algorithms in KWAL. For each read request, a corresponding redirection entry is searched from either the in-flight transaction table or the committed transaction table depending on the states of the file and inode (\textit{search()} function in Figure 3). If the redirection entry is found, a KWAL-redirected read request is sent to the native file system. Otherwise, KWAL delivers the read request to the native file system without redirection. Even when a relevant redirection entry exists in the committed transaction table, if the \textit{committed} flag of the entry is 0, the \textit{search()} function ignores the entry since the transaction commit of the block has been delayed for group commit.

For write requests, if the target file state is \textit{on-going} or \textit{need_CP}, a KWAL-redirected write request is sent to the native file system whether the redirection entry is found or not. If the redirection entry is not found, new free blocks allocated from the KWAL file via \textit{alloc_block()} function, and the redirected write is targeted for the new blocks. If the redirection entry is found, the old blocks are overwritten when the target file state is \textit{on-going}. However, if the target file state is \textit{need_CP}, the old committed blocks are invalidated and the new data are appended.

Even a non-transactional write must be redirected to the KWAL file if there are relevant committed blocks in the KWAL file. Otherwise, the data of the non-transactional write will be lost by the checkpointing of the KWAL file.

At the example in Figure 2, when process B sends a non-transactional read to the file \textit{foo} before the transaction commit of process A, KWAL delivers the read request to the native file system without redirection since there is no committed transaction of the file \textit{foo}. Therefore, process B reads the old data (D1) from the original file. When process A reads the same block, KWAL finds the redirection entry from the in-flight transaction table, and then sends a redirected read to the KWAL file. When process B reads the file \textit{foo} after the transaction commit of process A, KWAL finds the corresponding redirection entry from the committed transaction table, and redirects the read request to the KWAL file. Therefore, the committed data can be accessed by process B.

If \textit{tx_delayed_commit(fd_list)} is called, KWAL changes the transactions to the \textit{delayed} state. The KWAL metadata are not written at the KWAL file for the delayed transactions, and thus they are not crash recoverable. When \textit{tx_group_commit(fd_list)} is called, all the transactions in the \textit{fd_list} are committed atomically. KWAL changes the transactions to the \textit{committed} state, and moves the corresponding redirection entries in the in-flight transaction table to the committed transaction table. Therefore, the commit-delayed blocks can be accessed by other processes. The metadata of all pending transactions are merged and flushed at once. The group commit can be useful for synchronized transaction handling across multiple files and multiple processes. In addition, it can reduce the number of \textit{fdata-sync()} calls, proving a higher performance.

D. CHECKPOINT

KWAL checks the free space in the KWAL file at every \textit{tx_commit()} handling, and triggers the checkpoint operation if the free space is smaller than a certain threshold \(\theta_{cp}\). The value of \(\theta_{cp}\) must be determined to be sufficient to prevent overflow. The checkpoint operation is performed by a separate kernel thread, called \textit{CP thread}. While the checkpointing is processed by the CP thread, we need an additional KWAL file to support in-coming transactions. Therefore, KWAL maintains multiple KWAL files and uses them in rotation. Depending on the storage IO bandwidth, the checkpoint latency will be different. Considering IO bandwidth and transaction commit rate, a proper number of KWAL files must be prepared.

The detailed process of checkpoint operation is as follows: (1) The target inodes to be checkpointed are listed,
which are provided by the checkpoint inode list. At each transaction commit, the committed inode is inserted at the checkpoint inode list. (2) Per-inode read locks are set for the committed transaction tables of the target inodes to prevent further modifications, and the checkpoint redirection list is generated by referring to the transaction tables. If there are any open transactions for the target inode, the checkpoint thread waits until the transaction is closed (i.e., committed or aborted). Thus, it is guaranteed that all the transaction-related data of a transaction is recorded at only a single KWAL file.

(3) For each redirection entry of \((B_{\text{org}}, B_{\text{wal}})\) in the checkpoint redirection list, the KWAL file block at the address of \(B_{\text{wal}}\) is copied to the original file block at the address of \(B_{\text{org}}\).

(4) \(fsync()\) is called for each checkpointed target file.

(5) The checkpointed redirection entries are removed from the committed transaction tables, and the \text{need\_CP} flags are cleared. (6) The checkpointed KWAL file is initialized and changed to a spare KWAL file.

Since the checkpoint operation copies the logged data blocks to the target file, there are duplicated write operations, i.e., the same data are written twice to the KWAL file and the target file. The write amplification is harmful to the limited lifetime of flash memory storage such as solid-state disks (SSDs). In addition, the foreground transaction I/Os can interfere with the checkpoint operations.

To resolve this problem, we propose a new file system API, called \text{remap()}, which can make two files exchange their blocks logically by only changing the block mapping metadata of the files. If a native file system provides the block remap interface, KWAL can utilize it to remove the duplicated writes. If no such API is provided, KWAL can use the normal read and write operations to copy data blocks between the KWAL file and the target file. The \text{remap()} transfers three arguments: the source file descriptor, the destination file descriptor, and an array of remap entries. A remap entry contains the source block offset and the destination block offset. Whereas MapFS [26] proposed a similar block remap technique only for CoW-based file systems, our remap technique can be applied to non-CoW-based file systems such as EXT4 and F2FS. In addition, we considered the atomicity issue of the remap operation to utilize it for the transaction checkpointing.

We implemented the \text{remap()} function at four different file systems, EXT3, EXT4, XFS and F2FS. The \text{remap()} function modifies the file block address mappings in the target inodes (e.g., \text{i\_block} structure of EXT3/EXT4 and \text{i\_addr}/\text{i\_nid} structures of F2FS). Generally, native file systems can guarantee the atomicity of a single file API with their consistency-guaranteeing features such as journaling. However, even though the atomicity of \text{remap()} is not provided by native file systems, KWAL can know the progress of an interrupted remap operation by checking the checksums of target data blocks at crash recovery. Therefore, after a system crash occurs during a remap operation, replaying the checkpointing at crash recovery will not remap again the block which has been remapped by an interrupted checkpoint operation.

Since EXT4 and XFS use an extent-level mapping, one extent can be divided into several small sized extents after block remap. The fragmented extents can adversely affect the read and write latency. Since F2FS uses a log-structured write scheme, there is no fragmentation issue. To solve the fragmentation problem, we adopt a hybrid scheme, where either copy or remap operation is selected based on the state of the KWAL file to be checkpointed. We propose the \text{selective remap} technique as a hybrid approach. This technique measures two metrics of the target KWAL file:
validity ratio and average logical extent size. The validity ratio is the ratio of valid blocks in the KWAL file. A logical extent is a set of blocks whose block addresses are consecutive at both KWAL file and the original file. If the validity ratio is high, block remap will be more effective since many blocks must be copied in the copy-based scheme. When the average extent size is large, the fragmentation will not worsen under the remap-based scheme.

Our selective remap works as follows. If the validity ratio is larger than \texttt{FORCE_REMAP}, remap is used for checkpointing. If the validity ratio is between \texttt{FRAG_CHECK} and \texttt{FORCE_REMAP} (\texttt{FRAG_CHECK} < \texttt{FORCE_REMAP}), remap is used only when the average logical extent size is larger than \texttt{MIN_REMAP}. Otherwise, the copy-based scheme is used. The optimal values of \texttt{FRAG_CHECK}, \texttt{FORCE_REMAP} and \texttt{MIN_REMAP} will be different depending on the storage performance and IO workload. We determined the proper values by experiments.

A recent study of SplitFS [27] also proposed a similar API, called \texttt{relink()}, to perform block relocation between files by modifying only the metadata without any physical data movement. While SplitFS implemented the \texttt{relink} only in the EXT4 file system, we implemented remap for various file systems including EXT3, EXT4, XFS and F2FS, to demonstrate KWAL’s file system independence. Moreover, KWAL selectively uses copy or remap depending on the write workload to resolve the performance degradation by file fragmentation. We also considered the crash-consistency issue by the remap-based checkpointing (§II-F).

\section*{E. TRANSACTION ISOLATION SUPPORT}

1) \textbf{FINE-GRAINED WRITE CONFLICT DETECTION}

If two concurrent transactions update the same block of a file, a lost-update where the latter transaction overwrites the data of the first committed transaction can happen. The write conflict detection must check such a case and abort the latter transaction to ensure no lost-updates occur. Since KWAL manages block-level redirection entries, it is trivial to detect block-level write conflicts. However, because user write is byte-level, there is a \textit{false conflict} problem in which a block-level write conflict is incorrectly determined when two transactions modify different byte ranges of the same block. The transaction abort/retry due to the false conflict will cause performance degradation.

Unlike the previous techniques such as Isotope [7], KWAL can monitor the byte-level writes since it is located at the stackable file system layer. Considering a large amount of memory space required for byte-level write monitoring, KWAL is designed to support 64 byte-level write conflict detection by using an 8-byte sized dirty bitmap for each 4 KB block. The minimum granularity of write conflict detection can be adjusted considering the trade-off between performance and memory consumption.

KWAL first checks the block-level conflict during the transactional write handling, and then it performs the byte-level conflict detection only when the block-level conflict is detected. If a byte-level conflict is detected, the transaction is aborted. When there are no byte-level conflicts, the block-sharing transactions are merged at transaction commit. Figure 4(a) shows how block-level and byte-level conflicts are detected during the transactional write handling. KWAL uses a global dirty bitmap and a local dirty bitmap for 64B-level write conflict detection. The global dirty bitmap is located in the inode structure and maintains the information of byte regions modified by all block-level conflicting transactions.

After the redirection entries are added to the in-flight transaction table, KWAL records the local dirty bitmap for each modified block by referring to the parameters of write system call (①). KWAL registers the modified block numbers to the inode structure (②). At that time, KWAL can check for block-level write conflicts with other transactions by finding any duplicate from the pre-registered modified block numbers. The pointers to the corresponding redirection entries in the in-flight transaction table are also registered at the inode structure, which will be used to inform the transaction of any mergeable block-level conflict found by following other transactions. In the example of Figure 4(a), KWAL can know that the block number 0 has been already modified by the process P1. If there are conflicting blocks, KWAL compares the local bitmap in the file structure and the global bitmap in the inode structure (③). If there are overlapped bits between these two bitmaps, the current transaction is aborted since another transaction has already modified the same byte region. Otherwise, the local bitmap of the current transaction is ORed to the global bitmap to enable other transactions to check byte-level conflicts with the current transaction (④). KWAL sets the \texttt{conflict} flags (§II-B) of the redirection entries in the current transaction’s in-flight transaction table to denote that a byte-level transaction commit is required for the block (⑤). Then, KWAL also sets the \texttt{conflict} flag of the redirection entry in the transaction that caused the block-level conflict.

If \texttt{tx_abort()} is called, the local bitmap and global dirty bitmap information modified by the transaction must be rolled back. The global bitmap is XORed to the local bitmap of aborted transaction to roll back byte-level conflict information of the transaction. Figure 4(b) shows the byte-level transaction commit for transaction merge. First, only the 64B-level changes in the transaction data are extracted if the \texttt{conflict} flag is marked, and they are applied to the latest committed block, which can be found from either the original file or the KWAL file. The modified blocks are written at the KWAL file. Finally, KWAL clears the corresponding bits of the committed transaction in the global dirty bitmap.

2) \textbf{REPEATABLE READ ISOLATION}

KWAL supports the repeatable read isolation. An example of the violation of repeatable read isolation is as follows. Process B updates and commits a data block, which was already read by process A. When process A re-reads the block,
it will have a different value than the first read. If process A reads the block in a transaction, this is an isolation violation. KWAL ensures consistent data access, even in such a scenario. Without repeatable read isolation, the developer must use a lock on the file block for exclusive access. However, this approach leads to lower concurrency and lower transaction performance.

To support repeatable read isolation, KWAL allows multiple versions of committed blocks for the same original data block to exist if they have different commit times. KWAL maintains a per-inode virtual timer for each inode, which is incremented at every call of tx_start() and tx_commit() for the inode. When a process calls tx_start(), KWAL saves the current per-inode virtual time as the start time of the transaction in the KWAL file structure. When tx_commit() is called, the per-inode virtual time is written at the time fields of the corresponding redirection entries. In a transactional read, if a committed block has a later time than the start time of the current transaction, it is ignored by the transaction. For example, in the previous isolation violation scenario, when process A re-reads the shared block, the committed block modified by process B is ignored since its commit time is later than the transaction start time of process A, and an older version will be read by process A. The old redirection entries are removed from the committed transaction table if there are no relevant transactions on the inode.

F. RECOVERY

If the file system is unmounted normally, KWAL checkpoints any remaining KWAL file and unlinks it. Therefore, if any KWAL file is found during system startup, KWAL can know that crash recovery is required. At crash recovery, KWAL first reads the metadata of KWAL files. If there are multiple KWAL files, KWAL can check the order of them through the sequence number of each KWAL file. The KWAL file includes the metadata blocks of multiple committed transactions. For each transaction, KWAL checks the integrity of the corresponding metadata blocks with the recorded metadata checksum (II-B). If the checksum does not match the metadata blocks, the transaction is discarded since the metadata blocks are partially written. For valid metadata blocks, KWAL checks whether the data checksum of each redirection entry matches the corresponding KWAL data block. If the metadata of the transaction are completely stored but the relevant KWAL data blocks are not committed before a crash, the checksum will not match the KWAL data blocks. In such a case, the transaction is discarded. As a special case, the checksums may match the original data blocks. This means that the transaction is already checkpointed by the remap-based scheme. For valid committed transactions, KWAL replays the checkpointing.

In the copy-based checkpointing, the KWAL file is not modified after checkpointing. Therefore, KWAL can recover by simply re-copying the data blocks of KWAL file even when some blocks have already been copied. However, in the case of the remap-based scheme, replaying checkpoint operations can corrupt the original data blocks. Therefore, if the data checksum values in the KWAL metadata matches the original data blocks, the crash recovery skips the remap-based checkpointing for the corresponding blocks at crash recovery.

We conducted a virtual machine-based experiment used in [10] to test crash-consistency, and verified that KWAL provides crash-consistency through recovery in the event of system failure.

III. RELATED WORK

The previous kernel-level transactional file access techniques are as follows: TxFS [10], TxLFS [9], and F2FS atomic write [8] are studies to support user-defined transactions by leveraging the transaction mechanism provided by the file system. Since these studies depend on the implementation of the file system, they cannot be applied to various file systems. Moreover, the atomic writes supported by F2FS do not guarantee isolation, so they support only part of a transaction. Valor [1] is a study of transaction support in the VFS layer. It uses a separate log partition for WAL logging for transaction processing. Since Valor is implemented in the VFS layer, the existing kernel needs to be modified. Valor uses the lock-based transaction isolation. As a result, the multi-process transaction performance can be degraded. IsoTope [7] supports transactions by managing the entire storage as persistent logs at the block layer in the kernel. It cannot provide transactions for user-level file operations because it is in a lower layer than the file system. CFS [16] is a transactional file system leveraging a transactional storage, which includes X-FTL [20]. TxSSD [17] is the transaction-supporting SSD implemented at a real SATA-based SSD. TxOS [28] supports transactions for updates of memory buffers and data structures in kernel. It can support various system calls as
a transaction. However, this approach adds significant complexity to the kernel.

### IV. EVALUATION

For experiments, we used a system equipped with quad-core Intel i5-4570 CPU, 8 GB RAM, and Samsung 850 (256 GB) SSD. Linux kernel version was 4.9.11. Four KWAL files were pre-allocated and they were used in rotation. The size of a KWAL file is 256 MB. The KWAL checkpoint is triggered when the free space is less than 10 MB. The remap API was implemented in EXT4 and F2FS file systems. We modified the stackable *wrapfs* [29] to implement the KWAL file system.

We applied KWAL on EXT4, F2FS, and XFS to demonstrate its file system independence. The ordered-mode journaling was used at EXT4 at the experiments. We verified the correctness of KWAL by executing several benchmark applications including modified MySQL and SQLite, as will be shown in §IV-D and §IV-E. We used TxFS [10] for comparison, whose source codes are downloaded from https://github.com/ut-osa/txfs.

#### A. MICRO-BENCHMARK

To measure the overhead of KWAL in file IO operations, we used two simple workloads: overwrite and random read. These workloads generate write or read requests to a pre-existing 32 MB file. The IO request size is 16 KB or 64 KB. In the overwrite workload, *tx_commit()* or *fdatasync()* was called after each write request. We measured the IO bandwidth under different scenarios, native, TxFS, KWAL-notx, and KWAL-txIO. Whereas native uses the native EXT4 file system, KWAL-notx and KWAL-txIO uses the KWAL file system over EXT4. native and KWAL-notx generate non-transactional IOs and *fdatasync()* APIs. TxFS and KWAL-txIO use the transactional API to generate transactions.

Figure 5(a) shows the results of the overwrite workload. During the experiments, the checkpoint thread of KWAL was not invoked. There is no significant performance difference between KWAL-notx and native with the overwrite workload. Therefore, we can know that the overhead of KWAL file system is negligible with normal IOs. The transactional writes using KWAL (KWAL-txIO) shows a slightly lower performance than non-transactional writes with the sequential write workload, since KWAL must update the transaction tables for each transactional write. The transactional writes show 15.1% and 3.4% of performance degradation when the request sizes are 16 KB and 64 KB, respectively, with the sequential workload. However, the performance degradation is not serious. A similar experiment was conducted at [1], where Valor showed about 2.75 times performance degradation compared to the native file system. Therefore, we can say that the transaction processing overhead in KWAL is relatively low compared to Valor. Interestingly, the transactional writes outperform non-transactional writes with the random workload. In the KWAL scheme, all user write requests are sequentially appended to the KWAL file, regardless of user write patterns. Therefore, the random writes are transformed to sequential writes to the KWAL file. Because SSD devices have higher sequential write performance compared to random writes, KWAL can improve random workload performance. TxFS showed lower performance compared to KWAL at the overwrite workload. This is because TxFS has double write problem and JBD thread invocation overhead.

We also measured the random read performance under different scenarios. For the scenarios of KWAL-notx and KWAL-txIO, 16 MB of data were overwitten via transactional writes at the pre-existing 32 MB file as the initialization step. Since KWAL file size is 256 MB, no checkpoint operations occurred during the initialization. Then, we measured the read bandwidth running transactional or non-transactional random reads. The page caches were dropped before the measurement. Owing to the transactional writes at initialization, the committed transaction table in KWAL has many redirection entries, and both transactional read and non-transactional read must search for any relevant redirection entries from the transaction table. Therefore, we can measure the KWAL overhead for non-transactional reads. As shown in Figure 5(b), there is no significant performance difference among the four different scenarios. Therefore, we can know that the KWAL overhead for read request is negligible.

#### B. TRANSACTION ISOLATION

We implemented our own key-value store, called KWAL-KV, using KWAL APIs to demonstrate how easy it is to build a transactional application over KWAL. KWAL-KV is composed of an in-memory hash table to manage key-value pairs and a storage engine to support transactional IOs to DB files. KWAL-KV provides *get()* and *put()* as key-value APIs. The implementation of KWAL-KV was simple, requiring 370 lines of code. In particular, since KWAL supports write conflict detection and repeatable read isolation, no complex locking protocols were required to implement KWAL-KV. We verified the correctness of KWAL-KV by running the key-value benchmark of db_bench [30].

To observe the improvement by KWAL at transaction concurrency, we implemented two additional key-value stores: one uses the conflict avoidance scheme and another uses the transaction APIs of TxFS. The conflict avoidance scheme uses file-level locks at every access to the DB file to avoid read and write conflicts. TxFS supports block-level conflict detection, where a conflicted transaction is aborted and the

---

**Figure 5.** Micro-benchmark result.
transaction must restart from the beginning. In this experiment, the db_bench benchmark was used.

Figure 6(a) compares the performance of KWAL-KV with other key-value stores while varying the number of concurrent user threads. Each thread puts 10,000 random key-values, and each key-value pair consists of a 64B or 8B-sized key and a 4KB-sized value.

A block-level conflict occurs when two concurrent transactions attempt to update keys in the same block. Thus, as the number of threads increases, the number of block-level conflicts also increases. In the key-value store using the conflict avoidance scheme, even though the number of concurrent threads increases, the performance is not improved due to its file-level locking mechanism. However, KWAL-KV shows better performance as the thread concurrency increases. This is because KWAL uses 64B-level conflict detection. In particular, when the key size is 64 bytes, no byte-level conflicts occurred. Therefore, multiple threads process transactions concurrently, improving overall performance. Even when 8B-sized key is used and thus there are byte-level conflicts, the performance of KWAL is not degraded significantly. When there were four threads, the number of transaction restarts was only 7.2% of the total transactions in the 8B key experiment.

Since TxFS uses block-level write conflict detection, its performance slowed down as the number of threads increased. With two concurrently-running threads, transaction restarts occurred 3,375 times (12.6% of total transactions) in the TxFS-based key-value store. When there were three threads, the number of transaction restarts was increased to 36,218 (26.1%). In particular, when there were four threads, the experiment failed to complete.

We also verified the repeatable read isolation of KWAL using the example scenario in §II-E2. With the transactional API of KWAL, consistent data always could be read regardless of concurrent transactions.

C. GIT
Git is a widely-used version control system. The git add and git commit commands modify the index file, which is the application metadata for version management. To update the index file atomically, Git uses the atomic rename protocol. However, the current implementation of Git does not call fsync() after each system call for performance reasons [31]. Accordingly, if a system failure occurs while executing a git command, the index file can be truncated to zero bytes, resulting in data corruption [4]. To remedy this fault, we modified the Git implementation to be crash safe by calling fsync() after the temp file is updated or is renamed. Therefore, two fsync() calls are added for each index file update.

We compared the transaction handling times at different versions of Git implementations: the original Git, the modified Git using two fsync() calls, the TxFS-based implementation, and the KWAL-based implementation. In our experiment, we created a total of 1,000 empty files in the Git local repository, and called git add for each file creation. As shown in Figure 6(b), the KWAL-based Git shows only 14.1% of execution time overhead compared to the baseline, while providing crash consistency. The modified safe Git showed about twice the latency of KWAL-based Git, since KWAL calls only one fdatasync() for each transaction commit. TxFS-based Git also shows better performance than the modified Git, however, its performance is inferior to that of the KWAL scheme. This is because each transaction commit results in a journal flush in TxFS.

D. MySQL DOUBLE WRITE BUFFER
The MySQL storage engine takes a variant of journaling, called double write buffer (DWB), to deal with the partial page write problem. The multiple pages to be updated atomically are first appended to a separate journal area, i.e., the double write buffer. When all pages are written completely to the journal area, the pages are updated at their original DB files. The double write buffer inevitably causes duplicated writes. In addition, while updating the original file blocks, MySQL calls an fdatasync() for each DB file. Instead of using the DWB scheme, we modified MySQL to use the transactional writes of KWAL to update multiple database pages. The group commit feature of KWAL is utilized to avoid frequent fsync() calls.

For MySQL experiments, we ran the tpcc-mysql [32] benchmark workload, where one hundred warehouses were used, 30 connections were applied, and the running time was 20 minutes. The buffer pool of MySQL was configured to either 64 MB or 4 GB. As a native file system, EXT4 and F2FS were used. In the experiments using EXT4, we used three different transaction checkpoint techniques, i.e., copy, remap and selective remap, to observe the effect of file fragmentation. The threshold values of FORCE_REMAP, FRAG_CHECK, and MIN_REMAP in the selective remap were configured to 90%, 60%, and 1MB, respectively. The experiment of F2FS excluded the selective remap since there is no fragmentation issue. In the MySQL experiment, we could not compare KWAL with TxFS since TxFS failed to complete due to kernel error.

Figures 7(a) and 7(b) compare the transaction handling throughput (tpmC) of three different schemes, DWB, noDWB, and KWAL, when the native file systems are EXT4 and F2FS, respectively. With the noDWB configuration, the DB files are directly updated without using the DWB scheme. Thus, it shows higher performance than the DWB scheme but is unsafe. KWAL shows higher performance than the DWB
scheme by eliminating duplicate writes. KWAL also shows higher performance even compared to the noDWB scheme due to its group commit, which can reduce the number of fsync() calls.

The different checkpoint schemes of KWAL show different performances depending on the native file system and buffer pool size. When the native file system is EXT4, the remap-based checkpoint (KWAL-R) shows lower performance than the copy-based scheme (KWAL-C) if the buffer pool is small (64 MB). With a small buffer pool, the database pages will be flushed to the storage frequently. In particular, a single-page flush is invoked to load a missed database page. The single-page flushes generate random write requests, causing file fragmentation in the KWAL-R scheme. Moreover, since many pages are flushed and re-accessed repeatedly with a small buffer pool, a large amount of invalid blocks are generated in the KWAL file. Then, the copy-based scheme also has a low overhead at transaction checkpoint. As a result, the transaction performance is degraded compared to KWAL-C. In the case of F2FS, KWAL-R shows better performance than KWAL-C regardless of the buffer pool size.

With a large buffer pool (4 GB), MySQL will flush database pages only by periodic flush operations. MySQL maintains a buffer pool as an LRU list to evict only cold pages during the periodic flushes. Because a periodic flush updates multiple database files, the performance gain by group commit will be significant. The periodic flush evicts multiple sequential pages rather than small random pages, and thus KWAL-R will not suffer from fragmentation. Furthermore, since only cold pages are flushed, the KWAL file has a small amount of invalid data. As a result, KWAL-R can reduce the checkpoint latency compared to KWAL-C.

The selective remap scheme (KWAL-S) always shows higher performance regardless of buffer pool size by selecting the proper scheme based on the state of the KWAL file. In the selective remap scheme, almost all checkpoints used remap when the buffer pool was 4 GB, and about 25% of the checkpoints used remap when the buffer pool was 64 MB.

Figure 7(c) shows the average amount of write traffic per transaction in the tpcc-MySQL experiment. KWAL-C shows similar write traffics with the DBW scheme since it also generates duplicate writes. However, KWAL-C has better performance owing to fewer fsync() calls. The noDWB and KWAL-R schemes show about half the write traffic compared with the DBW scheme. Since KWAL-S uses remap or copy alternatively depending the workload, its write traffic is between those of KWAL-C and KWAL-R.

E. SQLite

SQLite [33] implements its transactional database file access using rollback journaling or write-ahead logging. We modified SQLite by replacing the user-level transaction schemes with simple KWAL APIs. To compare the performance of the KWAL-based SQLite and the legacy SQLite, we ran the tpcc workload [34], where one hundred warehouses were used and the running time was 10 minutes. Figure 7(d) shows the performance with different SQLite implementations. The rollback journaling (RBj) shows the lowest performance since it has a high overhead to create and delete a rollback journal file for each transaction commit. When the copy-based checkpoint is used, the KWAL-based implementation (KWAL-C) shows similar performance to the WAL-mode legacy SQLite. The experiments of TxFS [10] reported that TxFS showed similar performance to the WAL-mode SQLite. Therefore, we can say that KWAL-C and TxFS show similar performance at the SQLite workload.

However, if the remap-based checkpoint is used (KWAL-R), KWAL provides worse performance than the original SQLite due to fragmentation. In particular, since the tpcc workload generates random writes, the DB files and KWAL files are severely fragmented. However, with the selective remap technique, we can avoid the problem.

We observed the performance difference at different query types of tpcc, as shown in Figure 7(e). KWAL-C shows higher performance than the WAL-mode SQLite in the delivery (D), order_status (OS), and stock_level (SL) queries. These queries are read-intensive. The original WAL-mode SQLite maintains a hash table to access DB pages in a WAL file. During the execution of the tpcc workload, many entries of the hash table become invalid by DB page overwrites, but the invalid entries are not removed until transaction checkpoint to support repeatable read. Therefore, the search time at the hash table increases as the WAL file grows. However, KWAL can eliminate the old redirection entries immediately when the corresponding transaction is committed, while supporting repeatable read. As a result, KWAL shows better read performance. On the other hand, KWAL-C shows lower performance than the original WAL-mode SQLite in new_order (NO) and payment (P) queries, which are write-intensive workloads. This is because KWAL and the original SQLite uses different

---

**FIGURE 7.** Tpcc result.

---

D. Park, D. Shin: Stackable Transactional File System Using Kernel-Level WAL
metadata logging schemes. The original SQLite records the WAL metadata as a byte log in the WAL file. Therefore, the same block in the WAL file can be overwritten several times while logging metadata. So, it is crash-vulnerable if file systems or storage devices do not guarantee single block atomic update. Considering the problem, KWAL is designed to record the metadata of different transactions at different blocks in the KWAL file. Thus, the free space of KWAL file is consumed quickly compared to the original SQLite, and lower performance is shown due to frequent checkpoints.

V. CONCLUSION
We proposed a kernel-level transactional file access scheme, called KWAL. Compared to the previous transactional file systems, KWAL provides simple transaction APIs and has several distinct advantages such as file system independence, transaction isolation with high concurrency, and high performance. To demonstrate the usefulness of KWAL, we implemented a simple key-value store, fixed the wrong transaction implementation of the Git application, and simplified the implementation of the storage engines of MySQL and SQLite.

REFERENCES
[1] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, “Enabling transactional file access via lightweight kernel extensions,” in Proc. 7th USENIX Conf. File Storage Technol., 2009, pp. 29–42.
[2] J. Gray, “The transaction concept: Virtues and limitations,” in Proc. VLDB, vol. 81, 1981, pp. 144–154.
[3] S. Park, T. Kelly, and K. Shen, “Failure-atomic msgync(): A simple and efficient mechanism for preserving the integrity of durable data,” in Proc. 8th ACM Eur. Conf. Comput. Syst., 2013, pp. 225–238.
[4] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “All file systems are not created equal: On the complexity of crafting crash-consistent applications,” in Proc. 11th USENIX Symp. Operating Syst. Design Implement., 2014, pp. 433–448.
[5] M. A. Olson, K. Bostic, and M. I. Seltzer, “Berkeley DB,” in Proc. USENIX Annu. Tech. Conf., 1999, pp. 183–191.
[6] R. Sears, E. Brewer, E. Brewer, and E. Brewer, “Stasis: Flexible transactional storage,” in Proc. 7th Symp. Operating Syst. Design Implement., 2006, pp. 29–44.
[7] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherpoon, “Isotope: Transactional isolation for block storage,” in Proc. 14th USENIX Conf. File Storage Technol., 2016, pp. 23–37.
[8] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2FS: A new file system for flash storage,” in Proc. 13th USENIX Conf. File Storage Technol., 2015, pp. 273–286.
[9] M. I. Seltzer, “Transaction support in a log-structured file system,” in Proc. IEEE 9th Int. Conf. Data Eng., Apr. 1993, pp. 503–510.
[10] Y. Hu, Z. Zhu, I.Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel, “TxFS: Leveraging file-system crash consistency to provide ACID transactions,” ACM Trans. Storage, vol. 15, no. 2, pp. 1–20, Jun. 2019.
[11] O. Roehe, J. Bacin, and C. Mason, “BTRFS: The Linux B-tree filesystem,” ACM Trans. Storage, vol. 9, no. 3, pp. 31–32, Aug. 2013, Art. no. 9.
[12] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, “From ARIES to MARS: Transaction support for next-generation, solid-state drives,” in Proc. 24th ACM Symp. Operating Syst. Princ., Nov. 2013, pp. 197–212.
[13] V. Prabhakaran, T. L. Rodheffer, and L. Zhou, “Transactional flash,” in Proc. 8th USENIX Symp. Operating Syst. Design Implement., 2008, pp. 147–160.
[14] S. Verma, “Transaction support in the windows NTFS file system,” in Proc. 4th USENIX Conf. File Storage Technol. (FAST), 2005.
[15] F. Schmuck and J. Wylie, “Experience with transactions in QuickSilver,” ACM SIGOPS Operating Syst. Rev., vol. 25, no. 5, pp. 239–253, Oct. 1991.
[16] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, “Lightweight application-level crash consistency on transactional flash storage,” in Proc. USENIX Annu. Tech. Conf., 2015, pp. 221–234.
[17] Y. Son, H. Kang, J.-Y. Ha, J. Lee, H. Han, H. Jung, and H. Y. Yeom, “An empirical evaluation of enterprise and SATA-based transactional solid-state drives,” in Proc. IEEE 24th Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOTS), Sep. 2016, pp. 231–240.
[18] E. Zadok, “Writing stackable file systems,” Linux J., vol. 5, no. 109, pp. 22–25, 2003.
[19] H. J. Choi, S. H. Lim, and K. H. Park, “JFTL: A flash translation layer based on a journal remapping for flash memory,” ACM Trans. Storage, vol. 4, pp. 14:1–14:22, Jan. 2009.
[20] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, “X-FTL: Transactional FTL for SQLite databases,” in Proc. Int. Conf. Manage. Data, 2013, pp. 97–108.
[21] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “ANVI: Advanced virtualization for modern non-volatile memory devices,” in Proc. 15th USENIX Conf. File Storage Technol., 1995, pp. 111–118.
[22] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, “SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device,” in Proc. 15th USENIX Conf. File Storage Technol. (FAST), 2017, pp. 271–284.
[23] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson, “Improving SSD lifetime with byte-addressable metadata,” in Proc. Int. Symp. Memory Syst., Oct. 2017, pp. 374–384.
[24] K. Han and D. Shin, “Remap-based inter-partition copy for arrayed solid-state drives,” IEEE Trans. Comput., vol. 71, no. 7, pp. 1640–1654, Jul. 2022.
[25] J. N. Gray, R. A. Lorie, G. R. Putzola, and I. L. Traiger, Readings in Database Systems, San Mateo, CA, USA: Morgan Kaufmann, 1988.
[26] J. Wires, M. Spear, and A. Warfield, “Exposing file system mappings with MapFS,” in Proc. 3rd Workshop Hot Topics Storage File Syst., 2011 pp. 1–9.
[27] R. Karlebodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, “SplitFS: Reducing software overhead in file systems for persistent memory,” in Proc. 27th ACM Symp. Operating Syst. Princ., Oct. 2019, pp. 494–508.
[28] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, “Operating system transactions,” in Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ., 2009, pp. 161–176.
[29] E. Zadok, I. Badulescu, and A. Shin, “Extending file systems using stackable templates,” in Proc. USENIX Annu. Tech. Conf., Gen. Track, 1999, pp. 57–70.
[30] (2011). LevelIFR [Online]. Available: https://github.com/google/leveldb
[31] J. Corbet, “That massive filesystem thread,” Tech. Rep., 2009.
[32] (2011). TPCC-MySQL [Online]. Available: https://github.com/Parcona-Lab/tpcc-mysql
[33] (2000). SQLite [Online]. Available: https://www.sqlite.org
[34] (2011). TPCC-SQLite [Online]. Available: https://github.com/apache/tpcc

DAEJUN PARK received the Ph.D. degree in electrical and computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2020. He is currently a Senior Engineer with Samsung Electronics, South Korea. His research interests include embedded software, flash memory, file systems, and operating systems.

DONGKUN SHIN (Member, IEEE) received the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, South Korea, in 2004. From 2004 to 2007, he was a Senior Engineer with Samsung Electronics, South Korea. He is currently a Professor with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea. His research interests include embedded software, computer architecture, and operating systems.

IEEE Access