Fast and Failure-Consistent Updates of Application Data in Non-Volatile Main Memory File System

Jiaxin Ou and Jiwu Shu*
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
ojx11@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn

***Abstract*—Modern applications have their own update protocols to remain failure consistency. However, these protocols are implemented without a comprehensive understanding of the persistence properties of the underlying file systems and typically optimized for disk-based storage. As a result, they are complex, error-prone, and exhibit disappointing performance on emerging fast non-volatile memories (NVMs) due to excessive data copies.**

In this paper, we propose a novel file system optimized for non-volatile main memories (NVMMs), FCFS, that offers a series of easy-to-use file-based interfaces to enable both correctness and high performance for applications to consistently update their data on NVMM storage. Specifically, FCFS enables applications to atomically and selectively update multiple files efficiently on NVMM storage based on their consistency semantics. To this end, FCFS employs an *NVMM-optimized write-ahead logging (NoW)* mechanism to reduce the consistency cost by fully leveraging NVMM's byte addressability and high concurrency properties but without relying on the page-cache layer, which consists of two key techniques. First, *Hybrid Fine-grained Logging* decouples file system metadata log and data log to avoid the false sharing of log data and achieve a good tradeoff between the copy cost and log tracking overhead. Second, *Concurrently Selective Checkpointing* allows asynchronous checkpointing to be performed in parallel and with minimum data copies in order to improve its efficiency.

Evaluations on an emulated NVMM device demonstrate that FCFS's failure-consistent update protocol outperforms conventional protocols (i.e., write-ahead logging and shadow paging) by up to 276% and FCFS-based applications outperform the original ones by up to 93%. Importantly, FCFS requires modifications to no more than 0.06% of the original code for each ported application in order to support failure consistency.

I. Introduction

Emerging fast non-volatile memory (NVM) technologies, such as phase change memory (PCM), spin-torque transfer RAM (STT-RAM), and resistive RAM (ReRAM), offer non-volatility, high-speed, and byte-addressability. Placing NVM to the main memory bus will produce non-volatile main memory (NVMM), leading to memory-level performance for storage [1], [2]. Since many important applications, including text editors [3], relational databases [4], [5], and key-value stores [6], [7], are currently implemented atop file systems rather than directly on raw storage devices, the most straightforward way to use NVMM is building a file system on it in which legacy applications can run on top of the NVMM-based file system directly. Importantly, most of these applications need to update their data without risking corruption after a failure as failure-consistent update is a fundamental requirement for computer systems.

Nevertheless, existing NVMM-based file systems only provide mechanism for protecting their own metadata or data from corruption while ignoring the corresponding protection for application data, providing no consistent update protocols for application data [8], [9], [10], [11], [12], [13]. As a result, most applications shoulder the burden of implementing their own update protocols to achieve failure consistency. However, these update protocols are usually complex and error-prone as applications are unaware of the persistence properties of the underlying file systems [14], [15]. As an example, either reordered durability due to the existence of write reordering in the CPU caches or no guarantees for sector-granularity atomicity in existing NVMM-based file systems may cause applications to lose or corrupt data on system failures [14]. Moreover, most applications resort to the costly disk-optimized logging mechanism to support failure consistency [4], [6], [7], [5], leading to disappointing performance on emerging NVMM storage due to excessive data copies.

In this paper, we propose a novel file system optimized for non-volatile main memories, FCFS, that offers a series of easy-to-use file-based interfaces to enable both correctness and high performance for applications to consistently update their data on NVMM storage. Though write-ahead logging (WAL) [16], [17], [18], [19] and shadow paging [11] are the two dominate approaches to support failure consistency, blithely applying these techniques to ensure application's failure consistency in an NVMM-based file system can lead to disappointing performance due to the following reasons.

First, WAL first writes the data to the log area which we call *logging*. After the data has been completely written to the log area, it can be copied to the data area which is called *checkpointing*. While WAL works efficiently with hard disk drive (HDD) as writing to the log area sequentially avoids random disk access, this value is much lower for NVMM as random and sequential access of NVMM are nearly identical. In contrast, the overhead of double NVMM writes for every update in WAL significantly reduces the system performance due to the relatively long write latency of NVMM [1], [20].

Second, shadow paging performs an update to a new location. Once the new data is persistent, it modifies the reference to the old data to refer to the new data. However, it should perform recursive out-of-place updates of the pointer blocks up to the file system root to commit the updates. While the *Short-Circuit Shadow Paging (SCSP)* technique is an optimization

for NVMM so that commit can happen at any locations in the file system tree rather than must be propagated to the file system root by leveraging NVMM's byte addressability [11], this optimization is mainly beneficial to single and small atomic write operations which is ineffective for supporting the application-level consistency in most cases, as applications often require atomicity across multiple updates or files [21], where atomic operations may span a large portion of the file system tree, causing commit to occur at a common ancestor which still incurs a significant amount of extra data copies.

To address these problems, FCFS implements an NVMM-optimized WAL (NoW) scheme to support fast failure-consistent updates on NVMM storage. NoW addresses the challenges of how to efficiently use NVMM's byte addressability and high concurrency to enable fast and correct application's failure consistency in an NVMM-based file system.

Our proposed NoW scheme comprises **two key ideas**. First, *Hybrid Fine-grained Logging (HFL)* decouples the file system metadata log and data log to avoid the false sharing of log data and achieve a good tradeoff between the copy cost and log tracking overhead. To enable this, file system metadata log is organized to use byte-level undo logging to eliminate the high log tracking overhead. In contrast, *HFL* uses specialized redo logging at the cacheline granularity for file system data updates to achieve a good tradeoff between the copy cost and log tracking overhead. To help applications locate the latest uncheckpointed data in the redo log, FCFS further employs a *Two-Level Volatile Index (TLVI)* to track the uncheckpointed data versions at low cost in terms of both performance and space. Second, *Concurrently Selective Checkpointing (CSC)* allows committed updates to different data blocks to be checkpointed concurrently to enhance the concurrency of checkpointing, while committed updates of the same data block are carefully handled using the *Selective Checkpointing Algorithm* in order to ensure correctness and reduce unnecessary data copies, thereby significantly improving the efficiency of checkpointing. To ensure correct failure recovery due to out-of-order checkpointing, FCFS further carefully handles the log deallocation by maintaining two ordering properties.

This paper makes the following **contributions**.

- We are the first to design an NVMM-optimized file system, FCFS, which supports application's failure consistency. Specifically, FCFS provides some easy-to-use file-based interfaces so that applications can atomically update multiple files according to their consistency semantics. However, FCFS's consistency protocol does not rely on the page-cache layer.
- We propose *Hybrid Fine-grained Logging (HFL)* to avoid the false sharing of log data and achieve a good tradeoff between the copy cost and log tracking overhead by decoupling the file system metadata log and data log.
- We propose *Concurrently Selective Checkpointing (CSC)* to enhance the efficiency of checkpointing so as to improve the overall performance. This technique allows asynchronous checkpointing to be performed in parallel and with minimum data copies.
- We implement FCFS in Linux kernel 3.11.0. Evaluations on an emulated NVMM device demonstrate that FCFS's failure-consistent update protocol outperforms conventional protocols (i.e., write-ahead logging and shadow paging) by up to 276% and FCFS-based applications outperform the original ones by up to 93%. Importantly, FCFS requires modifications to no more than 0.06% of the original code for each ported application in order to support failure consistency.

II. Background and Motivation

A. Failure Consistency

We borrow the concept from transaction management in database systems to study the properties of failure consistency. In database management systems, a transaction typically has four properties: atomicity (A), consistency (C), isolation (I), and durability (D). To provide ACID properties, transaction management has two major component: (1) *concurrency control* to enable the execution of multiple transactions, and (2) *failure recovery* to allow systems to recover from unexpected failures [22], [23]. To distinguish from database-like ACID transactions, **failure consistency**, in this paper, mainly refers to *failure recovery*, which ensures the *atomicity* and *durability* of application data so that the system is able to recover to a consistent state from unexpected system failures. In contrast, isolation of concurrent execution of multiple transactions is the subject of concurrency control.

B. Motivation

To make applications become simpler, easier to develop, and more reliable, most prior transactional file systems [24], [25], [26], [27], [28], [21] are built on the page-cache based two-level architecture design, which are designed for block-based storage devices (e.g., hard disk or NAND flash). However, they are inefficient for NVMM storage due to the following two main reasons. First, redundant data copies between the DRAM page-cache and NVMM storage causes significant copy overhead. Second, the generic block layer also incurs high software stack overhead. Given the anticipated high performance characteristic of emerging NVMMs, recent research [10], [8] has reported that these two overheads can significantly degrade the NVMM system performance.

To avoid such overheads, state-of-the-art NVMM-optimized file systems [8], [9], [10], [11], [12] enable direct access to NVMM storage by eliminating the page-cache layer. At first glance, simply adopting the direct access policy may seem to be enough to provide fast application's failure-consistent updates in an NVMM-based file system using existing transaction mechanisms. However, this is not the case because existing transaction mechanisms, such as write-ahead logging, are optimized for disk-based storage, leading to suboptimal performance on NVMM storage due to excessive data copies.

To overcome this problem, we propose an NVMM-optimized WAL (NoW) scheme to support fast application's failure-consistent updates in an NVMM-based file system by

making full use of NVMM's byte addressability and high concurrency but without relying on the page-cache layer.

On one hand, direct access to NVMM storage allows system designers to use fine-grained logging [10], [17], [29] rather than block-based logging to reduce the logging cost of WAL. Moreover, asynchronous checkpointing [16] is also an efficient approach to improve the WAL performance by moving the checkpointing latency off the critical path under low storage load. On the other hand, these two dominate optimizations for the WAL technique are far from providing fast and correct application's failure-consistent updates in an NVMM-based file system because there are three major challenges that need to be overcome.

First, the logging granularity should be carefully selected so that a log unit will not be shared by different transactions in order to ensure correctness. Suppose that two transactions, `Tx1` and `Tx2`, are running. Assume that `Tx1` commits successfully and the system crashes before `Tx2` commits. To achieve failure consistency, all relevant log data of `Tx1` should be persisted to the file system during recovery while that of `Tx2` should not. However, suppose that a log unit is shared by `Tx1` and `Tx2`, then some inconsistent data of `Tx2` in this log unit will be falsely persisted to the file system during recovery, leading to inconsistency. As a result, avoiding the false sharing of log data is necessary in order to ensure correct failure recovery.

Second, there is a tradeoff between the copy cost and log tracking overhead which is affected by both the logging granularity and logging mode. Logging-based mechanism provides two alternative modes, which stores new data updates (redo logging mode) or old data values (undo logging mode) during the logging phase. In the redo logging mode, new data is written to the redo log area before the checkpointing phase during a transaction, all read operations have to first search the log area as the latest data may reside in the log area before being checkpointed to the file system. On one hand, redo logging allows the system to reduce the copy cost by performing checkpointing asynchronously and redesigning the checkpointing algorithm (see Section III-E). On the other hand, redo logging also introduces an additional tracking overhead for indexing the uncheckpointed data in the redo log area. More importantly, finer logging granularity causes larger log tracking overhead, posing a new challenge on deciding the appropriate logging granularity. On the contrary, undo logging first journals the old data and then performs in-place update to the file system for the new data, eliminating the tracking overhead for log data but requiring double writes for every update in the critical path.

Finally, simply performing checkpointing asynchronously does not work well for improving system performance under high storage load. Even with asynchronous checkpointing, checkpointing may still occur in the critical path with increasing data storage needs, because the foreground program threads may stall until the background checkpointing threads reclaim enough free log space. Therefore, enhancing the efficiency of checkpointing is still important in order to reduce the performance cost of checkpointing by making full use of the limited idle time to perform checkpointing, thereby boosting the overall system performance. Two main reasons limit the checkpointing performance: (1) serially replaying the centralized log in the strict commit order to ensure correctness limits the concurrency of checkpointing, which also inhibits the system scalability, and (2) checkpointing data to their original location constantly for every log block introduces large write amplification, significantly degrading the NVMM system performance.

III. FCFS Design

A. FCFS's Transactional Model

To avoid enforcing only one transactional model at the application level, FCFS provides a relaxed transactional model at the file system level, which gives application developers more freedom to achieve the required transactional model for their applications according to specific application semantic. Specifically, FCFS only guarantees the failure consistency of application data, while leaving the responsibility of isolation and concurrency control to the application level. As an example, if concurrent modifications to the same block or file should be avoided based on specific application semantic, the application developer should carefully handle this situation by using existing concurrency control primitives, such as file lock or mutex, in order to avoid data races. This is reasonable because different applications have different isolation semantics. While some applications (e.g., SQLite [4]) require strong isolation, others (e.g., Kyoto Cabinet [6]) have already relaxed their isolation level for performance optimization without compromising the correctness [21].

To be more specifically, failure-consistent updates of application data guarantees that a set of file system operations are completed in all or none fashion. In FCFS, we group this set of operations into an *AD-transaction* (as opposed to *ACID-transaction*), which ensures atomicity and durability. From the file system's view, an *AD-transaction* guarantees that "*all relevant metadata and data updates made by the transaction must be persisted atomically*". For the remainder of the paper, we refer to *AD-transaction* simply as *transaction*.

To make data durable on NVMM storage, we flush the data from CPU caches to NVMM using the `clflush`/`mfence` instructions, and assume that the `clflush` instruction guarantees that the flushing data actually reaches the durability point (i.e., NVMM device). While Intel has proposed new instructions (`CLWB`/ `CLFLUSHOPT`/ `PCOMMIT`) to improve the cacheline flush performance and CPU cache efficiency [30], they are still unavailable in existing hardware. This paper therefore does not take them into consideration.

B. FCFS's Transactional Interfaces

FCFS offers four new API calls to support application's transaction : (1) `tx_begin(TxInfo)` creates a new transaction; (2) `tx_add(TxID, Fd)` relates a file descriptor to a designated transaction after the transaction creation;

```
int fd1 = open(/file1, “O_RDWR”) ;
int fd2 = open(/file2, “O_RDWR”) ;
struct TxInfo { int num; int *fdSet;} ;
struct TxInfo info ;
info.num = 2 ;
info.fdSet = (int *) malloc(2 * sizeof(int)) ;
info.fdSet[0] = fd1, info.fdSet[1] = fd2 ;

// transaction begin
unsigned long TxID = tx_begin(&info) ;
write(fd1, “data1”) ;
write(fd2, “data2”) ;
tx_commit(TxID) ;  // commit the transaction
// transaction end

free(info.fdSet) ;
```

Fig. 1. Sample Code to use FCFS. *Code snippet that implements failure-consistent updates of two existing files with FCFS's transactional interfaces.*

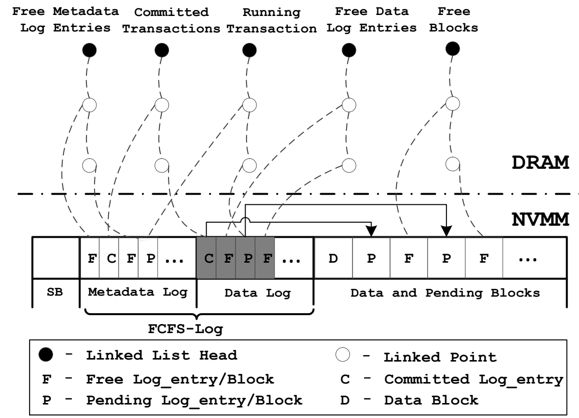
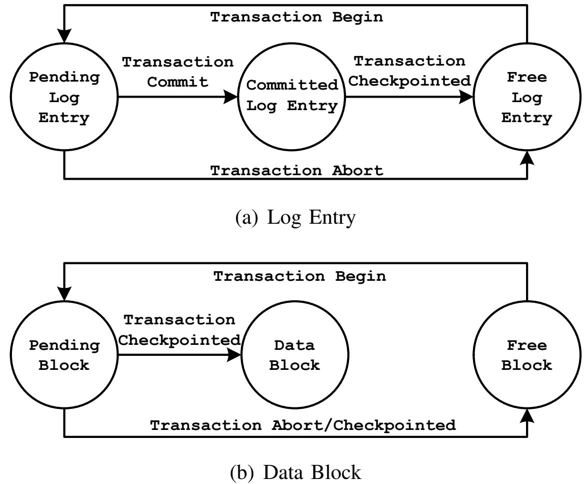


Fig. 2. FCFS Data Layout and Transaction Data Structures.



(a) Log Entry

(b) Data Block

Fig. 3. State Diagrams of Log Entry and Data Block in FCFS.

(3) tx_commit(TxID) commits a transaction; and (4) tx_abort(TxID) cancels a transaction entirely.

In FCFS, to process a set of file system operations atomically, applications only need to start a new transaction using the tx_begin() call before these operations and end the transaction with tx_commit() (or cancel it with tx_abort()) after them, rather than implementing complex consistent update protocols. To let the file system know which operations belong to a specified transaction, application developers should relate the corresponding file descriptors to the transaction, passing these information to the file system within the tx_begin() call or using the tx_add() call lazily after the transaction creation. After that, all related file operations before the end of the transaction belongs to this transaction. To finish the transaction, tx_commit() will make all related updates durable, and tx_abort() will undo all operations related to this transaction. Note that FCFS does not alter the existing file I/O interfaces within a transaction, which largely simplifies the use of its APIs.

Figure 1 shows a simple example in C of how to use FCFS to atomically update two files. The program first opens the two files. It then starts a new transaction using the tx_begin() call. The parameter info in tx_begin() specifies two file descriptors (fd1 and fd2) that belong to this transaction. Next, it performs two file write operations. It makes both write operations durable with the tx_commit() call. The atomicity that tx_commit() provides guarantees that either both writes become durable, or neither does.

C. Overview

The goal of FCFS is to enforce application's failure consistency at the file system level with both correctness and high performance. To this end, we propose an NVMM-optimized WAL (NoW) scheme, which consists of two key techniques.

- *Hybrid Fine-grained Logging (HFL)*, which decouples the file system metadata log and data log, to avoid the false sharing of log data and achieve a good tradeoff between the copy cost and log tracking overhead.
- *Concurrently Selective Checkpointing (CSC)*, where committed updates to different data blocks are checkpointed concurrently to enhance the concurrency of checkpointing, while committed updates of the same data block are carefully handled using the *Selective Checkpointing Algorithm* to ensure correctness and reduce unnecessary data copies.

FCFS Layout. FCFS data layout is shown in Figure 2. The superblock is followed by a journal (FCFS-Log) and dynamically allocated blocks. FCFS-Log is further divided into two areas, *metadata log* and *data log*, for different objectives. We will discuss the FCFS-Log layout in detail in Section III-D.

Allocator. Data allocation in FCFS is block-oriented, while journal allocation is based on the data structure (i.e., log entry). To avoid high logging and ordering overhead, FCFS maintains all the allocator structures in volatile memory using free lists rather than journaling them to the costly NVMM storage. The consistency of allocators will be discussed in Section III-F.

Creating and running transactions. When applications call tx_begin(), FCFS will create a new transaction and assign a transaction ID (TxID). In a running transaction, instead of directly overwriting the old-version data in the persistent data blocks, either old data (undo) or new data (redo) is journaled first to protect the old-version data. When a free log entry or block is allocated to this transaction, its state will change from *Free* to *Pending* (Figure 3). To ensure

correctness and achieve a good tradeoff between the copy cost and log tracking overhead, FCFS uses the *Hybrid Fine-grained Logging* technique, which decouples the file system metadata log and data log. To avoid the false sharing of log data and high log tracking overhead, FCFS uses byte-granularity undo logging for file system metadata updates in a transaction. In contrast, file system data updates are organized using specialized redo logging at the cacheline granularity, to avoid significant performance degradation caused by the costly double-write overhead in undo logging in the case of relatively large transactional data updates. In addition, FCFS keeps track of the cacheline-granularity data version in the redo log space using an efficient *Two-Level Volatile Index*. Details of FCFS's logging techniques will be discussed in Section III-D.

Committing/Aborting transactions. To commit a transaction, FCFS first persists all relevant metadata updates and data updates' logs (including the data log entries and the associated pending blocks) of this transaction to NVMM storage. It then appends a special commit log entry to the log area and make it durable, to indicate the completion of this transaction. When a transaction commits, the states of the log entries belonging to this transaction move from *Pending* to *Committed*. To abort a transaction, FCFS first undoes its relevant metadata log in the correct order, and then deallocates all the metadata log entries, data log entries, and pending blocks associated with it before making response to the application, to prevent conflicts and ensure forward progress. Aborting a transaction causes the states of its relevant log entries and pending blocks to change from *Pending* to *Free*.

Checkpointing transactions. Committed updates in the data redo log are periodically checkpointed to the file system. To improve the checkpointing performance, FCFS uses *Selective Checkpointing* in a *Concurrent* manner. To improve the concurrency of checkpointing by leveraging NVMM's high degree of parallelism, instead of serially checkpointing the committed log data in the strict commit order [16], committed modifications to different data blocks are checkpointed in parallel, while committed updates of the same data block are carefully handled using the *Selective Checkpointing Algorithm* to ensure correctness. In this algorithm, committed data is checkpointed to a *carefully-selected* block, which is selected from a set of relevant pending blocks and their common original data block, rather than to the original data block constantly, so as to reduce data copies. When checkpointing completes, the corresponding log space can be deallocated except the new permanent data block. We will discuss the details of FCFS's checkpointing technique in Section III-E.

D. Hybrid Fine-grained Logging

To avoid the false sharing of log data and achieve a good tradeoff between the copy cost and log tracking overhead, we propose the *Hybrid Fine-grained Logging (HFL)* technique, which decouples the file system metadata log and data log.

For file system metadata, the updates are typically small and the smallest unshared unit may be a single file system data structure (e.g., directory entry), which can be of arbitrary size. To avoid the false sharing of log data, the logging granularity for metadata updates should be byte. However, byte-granularity redo logging can significantly increase the log tracking overhead (e.g., every byte of log data may require at least 16 bytes of index data, 8 bytes for the index key and 8 bytes for the index value). To eliminate such overhead, the *HFL* technique uses byte-level undo logging for metadata updates so that the newest metadata is always written directly to the file system data area within a transaction.

In contrast, the same strategy cannot be efficiently applied to the data log as the size of data update may range from a few bytes to even several megabytes, depending on the access characteristics of specific workload. The drawback of double writes in the critical path for every update in undo logging can cause serious copy overhead and the associated write amplification, degrading system performance significantly in the case of relatively large transactional data updates. To balance this tradeoff, the *HFL* technique uses redo logging at the cacheline granularity for file system data updates. On one hand, redo logging offers a new opportunity to reduce the write amplification by redesigning the checkpointing algorithm (see Section III-E). On the other hand, cacheline-level logging also enables us to design a novel redo log index in order to efficiently search the log data, which we will discuss later. Note that though data can be accessed at arbitrary size, a cacheline block exclusively belongs to one file, thus no log data will be shared by different transactions as applications take the responsibility for ensuring that accesses of concurrent transactions to a single file must be ordered to avoid data races according to their semantics.

In a running transaction, instead of writing the journals to the volatile memory before commit, they are directly written to the FCFS-Log and the pending blocks on NVMM storage to avoid the data copy overhead between memory and storage. For file system metadata updates, FCFS first saves the old-version data by appending one or more log entries to the metadata log and making them durable before performing in-place updates to the file system. In contrast, file system data updates are logged to the newly-allocated pending blocks at the cacheline granularity, while their headers (i.e., journal metadata) are logged in the globally-visible data log area so that the pending blocks can be found during recovery. The decouple of log data and log header of the data log also offers a new opportunity for FCFS to support large transactions, as the size of transactional updates is mainly limited by the free space in the file system rather than a constant journal size.

To be able to identify the partially written log entries during recovery, FCFS includes a valid flag in each log entry. To remove the ordering between the persistence of the log entry and the valid flag, FCFS enforces that each log entry never crosses two separate cachelines, and leverages the architectural guarantee in the processor caching hierarchy that writes to the same cacheline are never reordered [10] to write a log entry. To achieve this, the valid flag is written last when writing a log entry to FCFS-Log, before the log entry is made durable. When deallocating a log entry, FCFS atomically set the valid

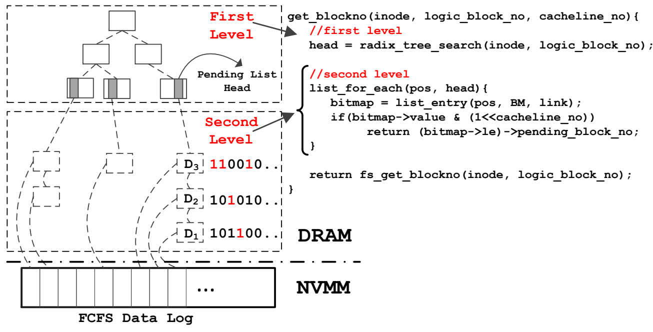


Fig. 4. Redo Log Index. D_1-D_3 *refer to different versions of data block* D*, the subscript digits represent the relative update order of this data block by different transactions. Function fs_get_blockno() will get the permanent block number of a logic block using the file system block mapping mechanism.*

flag of the log entry from 1 to 0 and make it durable by exploiting the processor feature for 8-byte atomic writes [11].

Redo Log Index. To help applications locate the latest uncheckpointed data in the pending blocks quickly, we should build an efficient index to search them. The simplest way to achieve this is to organize all the updated cachelines in the pending blocks using a tree structure or hash table. However, this approach incurs high overhead. As we discussed before, each uncheckpointed cacheline data (i.e., 64 bytes) requires at least 16 bytes of index data, demanding more than 25% of the total space of the pending blocks (including the space overhead of internal nodes). While limiting the log size by frequent checkpointing relieves this problem, it (1) increases the possibility of conflict between the background checkpointing threads and the foreground program processes, (2) decreases the opportunity of write coalescing of committed blocks across transactions during checkpointing, and (3) prevents large transaction support.

To reduce the index overhead while retaining the benefits of large log size, we propose a new technique, *Two-Level Volatile Index (TLVI)*, to support quick search to cachelines in the pending blocks. As shown in Figure 4, the *TLVI* technique organizes different update versions of a file system data block into a linked list, which we call *pending list*, in the commit order. Within each version's pending block, *TLVI* employs a cacheline bitmap to identify the updated cachelines during the transaction execution. In general, the *TLVI* technique consists of two indices to locate the block that contains the latest data of a specified cacheline: (1) the `first-level index` uses a per-file radix tree structure [31] to locate the pending list head of a data block, because there is a well implementation of the radix tree structure in the Linux kernel. To this end, all pending list heads of the data blocks of a file are organized into one radix tree structure and the root pointer of the radix tree is stored in each file's inode structure; and (2) the `second-level index` traverses the pending list from the newest version to the oldest one until the corresponding bit of its bitmap is 1, and then return its block number.

Take an example from Figure 4, the latest data of the $1st$ cacheline of data block D locates in the pending block D_3, while the latest data of the $3rd$ cacheline lies in the pending block D_2 because the corresponding bit in D_3's bitmap is 0.

The *TLVI* technique incurs low overhead in terms of both space and performance, while also providing the file system more opportunities to (1) perform checkpointing lazily during idle time, (2) benefit more from write coalescing in checkpointing, and (3) support large transactions.

From the perspective of space, for example, in the worst case, each 4 KB pending block requires a 16-bytes key-value pair for the first-level index and an 8-bytes bitmap for the second-level index, thus the log index requires less than 1% of the total space of the pending blocks. In addition, we expect that larger block size makes the *TLVI* technique even better. From the performance's view, the *TLVI* technique significantly reduces the space overhead of the index so that the index structure can be maintained in the small-but-fast volatile memory completely. However, the major challenge of the *TLVI* technique is how to limit the complexity of traversing the pending list in the second-level index. To resolve this, the concurrent checkpointing technique (see Section III-E) enables out-of-order checkpointing, which allows the background checkpointing threads to checkpoint the committed blocks in a pending list immediately as needed, rather than waiting for the strict commit order, as long as the size of the pending list is larger than a predefined threshold which is set to five blocks by default and is configurable.

E. Concurrently Selective Checkpointing

In FCFS, committed updates in the data redo log should be checkpointed to the file system data area eventually. During checkpointing, we must ensure that the process never applies the older version log data on top of a newer one, otherwise it leads to inconsistency. Common approaches for resolving this problem either use synchronous checkpointing [22] or employ asynchronous checkpointing [16] but perform it in the strict commit order. However, synchronous checkpointing forces the checkpointing latency to be always occurred in the critical path. In contrast, sequentially asynchronous checkpointing cannot fully utilize the high degree of parallelism of NVMM, which also inhibits the scalability. More importantly, both approaches constantly checkpoint the committed data to the original data block, incurring large write amplification.

In this section, we argue that enhancing the efficiency of asynchronous checkpointing is important, in order to reduce the performance cost of checkpointing by making full use of the limited idle time to perform checkpointing, especially with increasing data storage needs, thereby boosting the overall system performance. Towards this end, we propose a new technique, *Concurrently Selective Checkpointing (CSC)*, which allows asynchronous checkpointing to be performed in parallel and with minimum data copies.

In FCFS, checkpointing happens periodically when the number of free blocks drops below a predefined threshold (CHECKPOINT_THRESH), which is set to 10% of the total blocks by default. With asynchronous checkpointing, a data block may have different versions of pending blocks that are

committed in different transactions. While the committed updates to different data blocks can be checkpointed in parallel, different updates to the same data block should be handled carefully to ensure correctness. With *TLVI*, different versions' pending blocks of a data block have already been organized into a pending list. Therefore, the *CSC* technique processes the checkpointing of committed pending blocks across different pending lists concurrently using multiple threads, while checkpointing all committed pending blocks within a pending list in one group using the *Selective Checkpointing Algorithm*.

To ensure the correct failure recovery due to the out-of-order checkpointing, FCFS maintains two ordering properties during the log deallocation.

Ordering Property 1. *Different versions' redo log spaces of a data block should be deallocated in the correct commit order, including the redo log entries and their corresponding pending blocks.* Suppose that two different versions' redo log entries of a data block, L_1 and L_2 (L_2 is a newer version), finish the checkpointing process and begin to be deallocated. Assume that L_2 is deallocated successfully but the system crashes before deallocating L_1. Upon recovery, L_1 will be checkpointed to the file system while L_2 will not because it is already deallocated, which leads to inconsistency. The *CSC* technique resolves this issue by deallocating the log entries following the pending list order after the completion of checkpointing on this pending list, because the pending list has already ensured the correct commit order. Moreover, a redo log entry is deallocated before its associated pending block.

Ordering Property 2. *A commit log entry cannot be deallocated until all of its relevant undo and redo log spaces have been successfully deallocated.* Though the log entries for each transaction are written to the FCFS-Log in the non-consecutive manner, FCFS maintains a global committed list in the volatile memory (Figure 2), where the log entries for each committed transaction are linked within a consecutive list and the commit log entry is located at the end. When deallocating a log entry, its corresponding linked point will also be deleted from the global committed list. Thus, FCFS can deallocate a commit log entry only when its previous linked point in the global committed list does not point to a log entry that belongs to this transaction, because this situation indicates that all its relevant log entries have already been deallocated successfully. It is worth noting that all relevant undo log entries of a transaction can be deallocated immediately as long as it commits.

Selective Checkpointing Algorithm. To ensure correctness and reduce the copy overhead, FCFS checkpoints all committed pending blocks within a pending list in one group using the *Selective Checkpointing Algorithm* (Algorithm 1). The rationale behind this algorithm is that: To minimize the copy cost of checkpointing, a new permanent data block, which has the largest number of latest cachelines, is *carefully selected* among all committed pending blocks and the original permanent data block of this logic block (Step 1). Then, only the rest latest cacheline data, which does not lie in this new permanent data block, needs to be copied from other

Algorithm 1: Selective Checkpointing Algorithm

Input: ino: inode, lbn: logic_block_no

Output: no output, this function checkpoints all committed pending blocks of a logic data block.

```
1  selective_checkpointing (ino, lbn) begin
2      origin_pbn ← fs_get_blockno (ino, lbn) ;
3      // Step 1
4      new_pbn ← origin_pbn ;
5      max_lcn ← (the number of latest cachelines in origin_pbn) ;
6      for (each committed pending block cbn belonging to lbn) begin
7          if ((the number of latest cachelines in cbn) > max_lcn) then
8              new_pbn ← cbn ;
9              max_lcn ← (the number of latest cachelines in cbn) ;
10     // Step 2
11     for (each committed pending block cbn belonging to lbn) begin
12         if cbn ≠ new_pbn then
13             Copy the latest cacheline data from cbn to new_pbn and
               make it durable ;
14     if origin_pbn ≠ new_pbn then
15         Copy the latest cacheline data from origin_pbn to new_pbn
           and make it durable ;
16     // Step 3
17     if origin_pbn ≠ new_pbn then
18         Modify the reference to origin_pbn to refer to new_pbn and
           make it durable ;
19     // Step 4
20     for (each committed pending block cbn belonging to lbn) begin
21         if cbn ≠ new_pbn then
22             Deallocate cbn ;
23     if origin_pbn ≠ new_pbn then
24         Deallocate origin_pbn ;
```

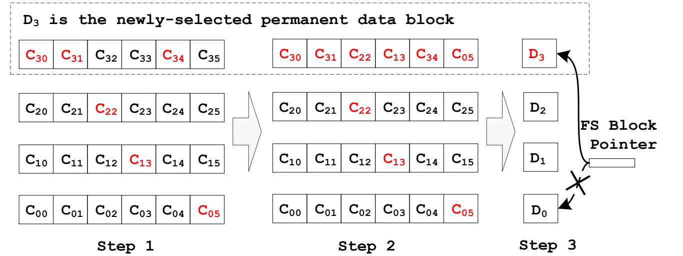


Fig. 5. A Simple Example of the Selective Checkpointing Algorithm. D_0-D_3 *refer to different versions of data block D, among which* D_0 *is the original permanent data block and* D_1-D_3 *are the committed pending blocks.* C_{ij} *means the jth cacheline in the ith version of block D. The red one indicates the latest cacheline of block D located in each version's data block.*

committed pending blocks and the original permanent data block to this block based on their cacheline bitmaps and made durable (Step 2). Finally, only an additional 8-bytes file system block pointer may need to be atomically modified and made durable to redirect from the original permanent data block to the new permanent data block (Step 3).

Figure 5 shows a simple example of the *Selective Checkpointing Algorithm*. Suppose that each block consists of six cachelines and each cacheline is 64 bytes. In the first step, D_3 is selected as the new permanent data block because it contains the largest number of latest cachelines. Then, cacheline data in C_{22}, C_{13}, and C_{05} is copied from D_2, D_1, and D_0 to D_3, respectively, in the second step. Finally, we modify the file

system block pointer to D_0 to refer to D_3, and deallocate other blocks (i.e., D_0-D_2). In the conventional WAL scheme, the checkpointing program needs to copy all cacheline data from D_1-D_3 to D_0 respectively, which generates 1152-bytes data copies. By contrast, FCFS's *Selective Checkpointing Algorithm* generates only three 64-bytes cacheline data copies and one 8-bytes pointer modification, which equals to only 200-bytes data copies, significantly reducing the copy cost of checkpointing by fully leveraging NVMM's byte-addressability.

F. Recovery

Fast recovery is important to provide high system availability and avoid data loss in the event of a failure. On restart after an unexpected failure, each log entry in the FCFS-Log area has one of the three statues: *invalid (free)*, *uncommitted (pending)*, and *committed*. To provide fast recovery, FCFS only undoes the uncommitted metadata log entires to revert the effects of any uncommitted transactions, while delaying the checkpointing of committed data log entries to amortize the cost of checkpointing by rebuilding necessary DRAM data structures for uncheckpointed transactions.

Recovery steps are as follows:

(1) *Building an Index for All Committed TxIDs*. All the log entries in the FCFS-Log area will be scanned in the first step. The type of each log entry (i.e., metadata log entry, data log entry, or commit log entry) is identified using the special *type* field in each log entry. The goal of the first step is to identify all valid commit log entries and then build an efficient volatile index (e.g., hash table) for all of the committed TxIDs.

(2) *Rebuilding DRAM Data Structures for Uncheckpointed Redo Logs.* With the *tx_id* field in each log entry and the index of the committed TxIDs built in the first step, the recovery process extracts all committed data log entries and invalids all uncommitted data log entries by scanning the data log area in the second step. Then, it sorts all committed data log entries by the TxID using sort algorithms to indicate the correct commit sequence. Finally, these committed data log entries are added to the global committed list and the redo log index one by one following the commit sequence.

(3) *Applying Uncommitted Metadata Undo Logs.* In the final step, all uncommitted metadata log entries can be detected similar to the second step. Unlike redo log, uncommitted undo log entries in the same transaction should be reapplied in the reverse commit sequence, because the same metadata may have been updated multiple times in a transaction and each update has an independent undo log record. To address this issue, each metadata log entry has an extra *sequence_id* field to identify the different versions of the same metadata in a transaction. Then, all uncommitted metadata log entries of the same transaction are sorted by the *sequence_id*. At last, they are reapplied in the reverse commit sequence so that the foremost consistent value can be applied to the file system, while the committed metadata log entries are invalided.

After the above three steps, while some committed data has not been checkpointed to the file system, the newest versions of the uncheckpointed data can be found using the redo log

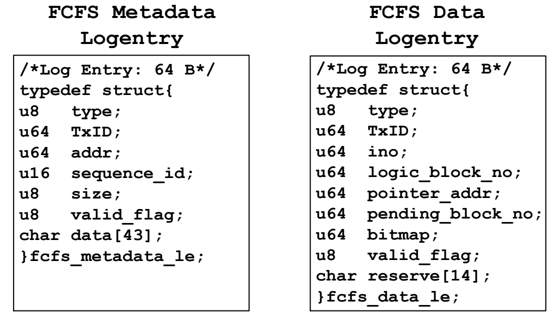


Fig. 6. FCFS Metadata and Data Log Entry Structures.

index, and the checkpointing can be performed lazily using the global committed list and the pending lists in the redo log index. Also, the effects of any uncommitted metadata log entries have been reverted by applying them to the file system in the correct order. As such, the data is guaranteed to be consistent.

After correct recovery, FCFS ensures the allocator consistency by walking the file system metadata to rebuild the data allocator structure and adding all invalid log entries to the corresponding free lists to rebuild the journal allocator structures.

G. Legacy Application Support

To be able to support legacy applications without any modifications, FCFS treats every system call from legacy applications to the file system as an independent transaction if its corresponding file has not been related to any transactions. To this end, such system call (e.g., a *write* operation) will be assigned a new transaction ID and committed automatically by the file system before making respond to the applications. At this time, FCFS provides a special semantic which resembles to the journal data mode in the conventional journaling file system (e.g., Ext4 in `data_journal` mode).

IV. Implementation

FCFS is implemented based on the PMFS [10] file system in Linux kernel 3.11.0, which modifies 5,212 lines of the original code. FCFS shares the file system data structures with PMFS but adds FCFS's transactional interfaces using new syscalls and the corresponding failure-consistent update protocol.

Figure 6 shows the structures of FCFS-based log entries. For a metadata log entry, the log data is stored along with the log head. We extend PMFS's log entry structure to implement FCFS's metadata undo log entry structure, in which the extra $sequence_id$ field is added to indicate the metadata update sequence within a transaction. Different from the metadata log entry, the data log entry in FCFS only comprises the log head, which means the log data is stored in the separate space from the log head. As a result, the $pending_block_no$ points to the pending block which stores the actual log data of this data log entry. Moreover, the ino (i.e., file inode number) and the $logic_block_no$ fields are used for the recovery program to add a valid committed log entry to a specified redo log index because each file has an independent index structure.

To support the *Selective Checkpointing Algorithm*, FCFS also needs to journal the file system block pointer address (i.e., $pointer_addr$) where it stores the block pointer to the file system permanent data block of this logic block. Finally, the $reserve$ array field does not store valid data but guarantees that the size of a data log entry equals to a cacheline size so that a log entry never crosses two separate cachelines.

V. Application Case Studies

We have ported three real-world applications to FCFS to illustrate how applications can use FCFS's transactional interfaces. The applications include the SQLite [4] relational database, the MySQL [5] relational database, and the Kyoto Cabinet [6] key-value store. To port these applications to FCFS in order to simplify the complex failure-consistent update protocols of existing applications, we reuse the original concurrency control code in each ported application, while replacing its failure-consistent update protocol with FCFS-based transactional interfaces, so that the original isolation level that is provided by each application is not compromised.

A. SQLite

SQLite [4] is an embedded SQL database engine which is widely used in smartphones and mobile computing field. To support the atomicity of transaction execution (i.e., failure consistency), SQLite provides two optional approaches, including *rollback journaling (RBJ)* [32] and *write-ahead logging (W-AL)* [33], for users to choose flexibly. In the RBJ mode, SQLite will write a copy of the original unchanged database content into a separate rollback journal file before writing changes into the database file, so that the uncommitted changes can always be rolled back. However, the WAL approach inverts this, in which the original content is preserved in the database file and the changes are appended into a separate WAL file, so that any committed changes can always be redone.

We have ported SQLite-3.8.11 to FCFS by replacing the RBJ and WAL approaches with FCFS-based transactional interfaces while ensuring the same level of failure consistency as them. Surprisingly, FCFS's change requires modifications to just 53 lines of code (LoC), or 0.03% of the original SQLite source code (169,486 LoC).

B. MySQL

We use an open-source implementation of the MySQL database, namely MariaDB [5], as one of our application case studies. MariaDB is an enhanced, drop-in replacement for MySQL which is widely used. Moreover, it comprises a rich ecosystem of storage engines, among which InnoDB [34] is a popular transactional storage engine. To support failure consistency, InnoDB employs an interesting technique called *double-write* [35], which means it will write data twice when it performs table space writes.

We replace the *double-write* mechanism that is used by original MariaDB's InnoDB engine with FCFS-based transactional interfaces to port the MariaDB-10.0.16 database to FCFS, which requires modifications to only 289 LoC, or 0.02% of its original source code (1,366,215 LoC).

C. Kyoto Cabinet

Kyoto Cabinet [6] is a library-based key-value store. Unlike previous relational databases, it employs memory-mapped I/O rather than file-based I/O to store the database content into the underlying file system. Moreover, it supports the failure consistency by using the write-ahead logging mechanism [36].

To port Kyotocabinet-1.2.76 to FCFS in order to replace its write-ahead logging mechanism with FCFS's failure-consistent update protocol, we first alter all the database I/O operations from using the memory-mapped I/O interface (i.e., *memcpy()*) to use the file-based I/O interfaces (i.e., *read* and *write* system calls), and then add the corresponding FCFS-based transaction begin (i.e., *tx_begin()*) and transaction end (i.e., *tx_commit()* and *tx_abort()*) functions at the correct locations. The porting efforts of this application requires changes to just 44 LoC, or 0.06% of its original code (73,945 LoC).

VI. Evaluation

In this section, we present the evaluation results of FCFS and answer the following questions:

(1) Does FCFS really preserve the consistency of application data? (Section VI-B)
(2) How does FCFS's failure-consistent update protocol compare with existing protocols in terms of performance? (Section VI-C)
(3) How sensitive are FCFS's failure-consistent updates to the variation of the transaction value size, the transaction idle time, and the NVMM write latency? (Section VI-D)
(4) What is the failure-recovery performance of FCFS-based protocol? (Section VI-E)
(5) How many performance benefits real-applications can obtain from FCFS? (Section VI-F)

A. Experimental Setup

Because real NVMM device is not available for us yet, we develop an NVMM performance emulator based on the NVMM emulator used in Mnemosyne [16] to evaluate our system. Our emulator uses DRAM to emulate NVMM and emulates both the NVMM write latency and the NVMM write bandwidth. The NVMM write latency is emulated by introducing an extra configurable software delay after executing the *clflush* instruction, while the NVMM write bandwidth is emulated by limiting the maximum number of concurrent NVMM writing threads to $(B_{NVMM}/(1/L_{NVMM}))$, where B_{NVMM} indicates NVMM's write bandwidth and L_{NVMM} is NVMM's write latency. Any overflow writing threads are queued which will be woken up when one of the current writing threads completes. To compare with conventional transactional file system which is based on the block-based file system, we also construct an NVMM-based block device emulator (NVMMBD) by modifying Linux's RAM disk module (`brd` device driver) and using the above NVMM performance model to emulate the NVMM latency and bandwidth.

All the experiments are conducted on a x86_64 Linux 3.11.0 kernel machine which is configured with 2.1 GHz Intel Xeon E5-2620 twelve-core processor and 16 GB physical

memory. Unless otherwise specified, we emulate NVMM by setting the write latency and write bandwidth to 150 ns and 4 GB/s respectively, the configuration used in the Mnemosyne project [16]. For all the experiments, each data-point is calculated using the average of at least 5 executions.

B. Correctness

FCFS should protect the consistency of application data upon failures. However, the NVMM emulator does not actually guarantee the data durability upon system failures because of the volatility property of DRAM. As a result, we cannot inject any real system failures (e.g., power interruptions or kernel crashes) for the correctness evaluation of FCFS. To simulate system failures as real as possible in order to verify the correctness of FCFS, we inject two types of user-level process failures into the FCFS-based MySQL database: *failure points* and *random process interruptions*. Upon these user process failures, we then unmount the FCFS file system without cleaning the FCFS-Log space to simulate the system failures. Finally, we mount the file system again to force failure recovery, and then check the consistency of the database using its own consistency checker tool to see whether FCFS can preserve the integrity of application data.

First, we manually insert several failure points into the FCFS-based MySQL database source code where we believe are most likely to cause inconsistency (e.g., before or after transaction commits). The result of triggering one of the failure points in the database source code is that the running user process will be terminated immediately. Then, we complement the correctness evaluation by randomly interrupting the user process when running the Sysbench [37] benchmark on the FCFS-based MySQL database to further simulate sudden failures. Upon restarting the file system from these process failures, if `mysqlcheck` [38] passes, then we conclude that FCFS is failure-consistent for application data updates. In our evaluations, we find that FCFS successfully passes the `mysqlcheck` in all failure simulation tests, including 18 failure-point tests and 200 random process interruptions. Therefore, we conclude that FCFS preserves the consistency of application data.

C. Overall Performance of Failure-Consistent Updates

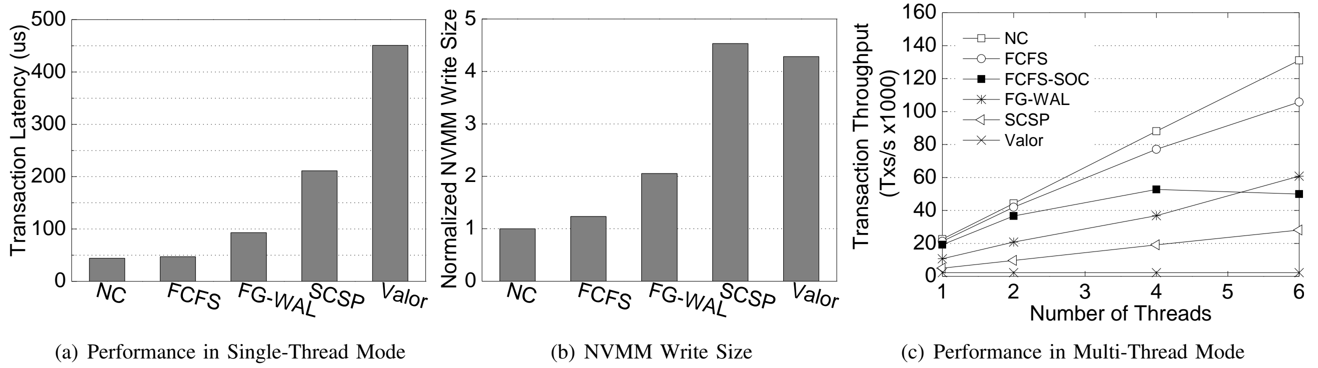
To understand the performance benefits of FCFS-based failure-consistent update protocol, we measure the performance when running the microbenchmark of atomically overwriting two existing files as shown in Figure 1. We compare the FCFS-based failure-consistent update protocol with three existing protocols (i.e., `FG-WAL`, `SCSP`, and `Valor`) and a no-consistency (i.e., `NC`) system. We use PMFS [10] to represent the `NC` system which has no guarantees for application's failure consistency. For a fair comparison, we choose two representative and optimized implementation of the existing failure-consistent update protocols. The `FG-WAL` system implements the failure-consistent update protocol in the `NC` system using an optimized WAL scheme, namely fine-grained (i.e., cacheline-level) write-ahead logging technique [10] for file system data updates. The `SCSP` system implements the failure-consistent update protocol in the `NC` system using an optimized shadow paging scheme, namely short-circuit shadow paging technique [11]. Finally, the `Valor` system is an userspace implementation of the essential part of Valor [25], which is built on the NVMMBD emulator and indicates the conventional page-cache based transactional file system.

Microbenchmark: For each experiment, each running thread performs 500,000 transactions. Inside each transaction, it randomly chooses two files from the 1,000 existing 4 MB files and performs two random 0-16 KB write calls in the two files respectively. We set the NVMM size to 6 GB and the background checkpointing process is triggered when there are less than 10% free blocks.

1) Single-thread Evaluation: To focus on the consistency effect, we first run the microbenchmark in a single thread to prevent the effect of concurrency control. Figure 7(a) shows the transaction latency (i.e., average execution time of a transaction) of the afore-mentioned systems when running the microbenchmark. As shown in this figure, the latency of FCFS-based version is the lowest among all failure-consistent versions - 49%, 78%, and 90% lower than the FG-WAL, SCSP, and Valor system respectively, and it is only 6% higher than the NC system. The source of this enhancement mainly comes from three aspects: the fast data redo log index, the high performance of the checkpointing process due to the concurrent execution and low data copy overhead of checkpointing, and the elimination of the page-cache layer. As expected, the NVMM write sizes of the three existing failure-consistent protocols are surprisingly high as shown in Figure 7(b) - 67%, 267%, and 247% higher than the FCFS-based version for the FG-WAL, SCSP, and Valor system respectively. Moreover, the Valor system has comparative NVMM write size with the SCSP system but introduces the page-cache layer overhead, thereby having much higher latency than it.

2) Multi-thread Evaluation: To further evaluate the performance with both consistency and concurrency control effects, we run the microbenchmark in multiple threads and vary the number of threads. To show the benefits of the concurrent execution of checkpointing, we also compare FCFS with `FCFS-SOC`, which performs selective checkpointing for each pending block asynchronously but in the strict commit order. The result is shown in Figure 7(c).

Figure 7(c) shows the transaction throughput (i.e., average transactions per second). As shown in this figure, the performance of Valor remains constant as the thread count increases due to the strong isolation support only. In contrast, the transaction throughput of FCFS increases as the number of threads increases, demonstrating the benefits of relaxing the isolation. However, the performance of FCFS-SOC remains nearly constant when the thread count goes from 4 to 6. This is because the transaction performance is constrained by the strict-order checkpointing technique. In this figure, we also observe that the performance gap between the FCFS system and the NC system becomes large as the thread count increases. This is because the background checkpointing threads in



(a) Performance in Single-Thread Mode (b) NVMM Write Size (c) Performance in Multi-Thread Mode

Fig. 7. Overall Failure-Consistent Updates Evaluations using Microbenchmarks. *The NVMM write size is normalized to that of the NC system.*

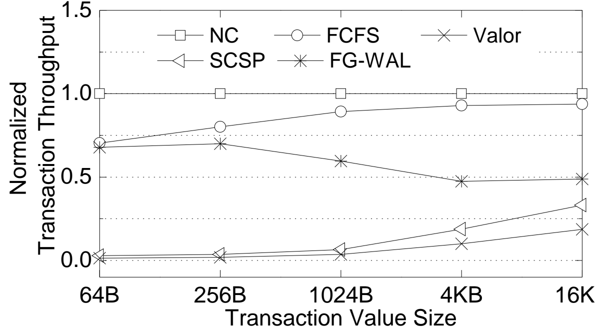


Fig. 8. Sensitivity to the Transaction Value Size. *Normalized to the throughput of the NC system.*

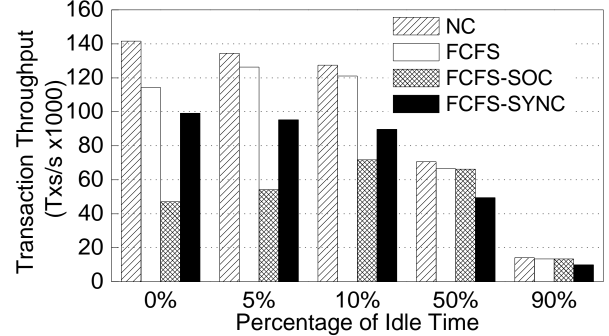


Fig. 9. Sensitivity to the Transaction Idle Time.

FCFS can hardly influence the performance of the foreground processes when there is surplus NVMM write bandwidth in the case of small thread count, while its performance is mainly determined by the NVMM write size factor under large thread count. Even with 6 threads, however, the FCFS-based version still outperforms FG-WAL and SCSP by 74% and 276% respectively due to the reduced NVMM write size.

D. Sensitivity Analysis

1) Sensitivity to the Transaction Value Size: We measure the transaction throughput by varying the transaction value size in the file write calls from 64 B to 16 KB, rather than choosing it randomly, when running the previous microbenchmark.

Figure 8 shows the normalized transaction throughput with different value sizes. From this figure, we observe that, as the value size increases, the performance improvement of FCFS over FG-WAL increases, while the performance degradation of FCFS compared to NC decreases. For instance, the FCFS-based system outperforms the FG-WAL system by only 3.5% in the case of 64 B value size, but improves the performance by more than 90% over FG-WAL when the value size is no less than 4 KB. The main reason is that the storage overhead is less significant than the software overhead (e.g., system call, user-kernel mode switch, and the additional redo log index) when the I/O size is small in the NVMM system. Conversely, it can dominate the NVMM system performance degradation when the I/O size becomes larger. We conclude that the price FCFS pays for failure-consistency is modest for relatively large transactional updates. In other words, FCFS gains more benefits in workloads with larger transaction value sizes.

2) Sensitivity to the Transaction Idle Time: To demonstrate that FCFS can make full use of the limited idle time to perform checkpointing thereby minimizing the consistency cost, we run the previous microbenchmark and vary the percentage of the transaction idle time from 0% (no idle time) to 90%. Moreover, we compare FCFS with NC, FCFS-SOC, and FCFS-SYNC. `FCFS-SYNC` implements a synchronous checkpointing technique, which checkpoints the pending blocks to the file system during the transaction commit phase using the *Selective Checkpointing Algorithm*.

Figure 9 shows the transaction throughput of all evaluated systems with different percentages of the idle time. We observe that, when the percentage of the idle time is no less than 50%, both FCFS and FCFS-SOC have better performance than FCFS-SYNC, and they outperform FCFS-SYNC by 34-36%. This is mainly because the idle time is large enough so that the background checkpointing speed can keep up with foreground transaction execution speed even though the checkpointing is performed in the strict commit order. However, with no more than 10% idle time, FCFS-SOC underperforms FCFS-SYNC by up to 53% (0% idle time) due to that the performance of FCFS-SOC is constrained by the strict-order checkpointing technique. In contrast, the FCFS-based protocol enables concurrent checkpointing by making full use of the NVMM bandwidth which eliminates such bottleneck, thereby always showing better performance than both FCFS-SOC and FCFS-SYNC even in the case of small transaction idle time.

3) Sensitivity to the NVMM Write Latency: Figure 10 shows the transaction throughput performance when we vary the NVMM write latency from 50 ns to 1600 ns running

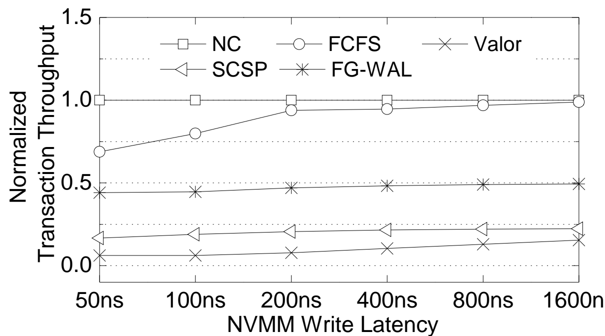


Fig. 10. Sensitivity to the NVMM Write Latency. *Normalized to the throughput of the NC system.*

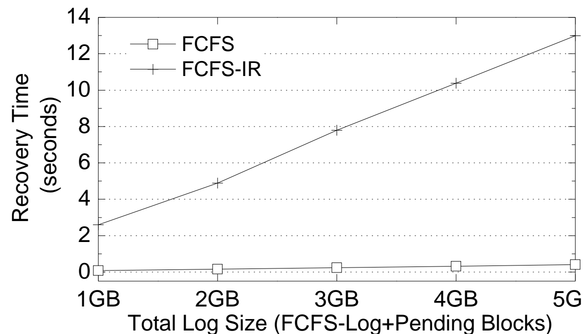


Fig. 11. Impact of Log Size on the Recovery Time. *The log includes the FCFS-Log and the associated pending blocks.*

previous microbenchmark. In this figure, we observe that the relative performance of FCFS to the NC system increases as the NVMM write latency becomes longer. The main reason is that, when the write latency of NVMM is close to that of DRAM, the overhead of the redo log index (i.e., the *TLVI* technique) can affect FCFS's performance to some extent even though the index structures are located in DRAM. In contrast, when the write latency of NVMM is much larger than that of DRAM, this overhead can be ignored because the large performance gap between DRAM and NVMM decides that all index accesses only account for a small portion of the total overheads.

E. Recovery

To study the impact of the total log size (i.e., the size of FCFS-Log and Pending Blocks) on the recovery time, we change the CHECKPOINT_THRESH, the frequency of checkpointing the pending blocks, to vary the total log size from 1 GB to 5 GB to measure its implication on the recovery time. To demonstrate the benefits of FCFS's recovery mechanism, we compare the recovery time of FCFS and FCFS-IR. `FCFS-IR` is a version of FCFS which checkpoints all committed transactions instantly during the recovery phase.

Figure 11 shows the recovery time of FCFS and FCFS-IR for different log size configurations. In this figure, we observe that, although both FCFS's and FCFS-IR's recovery time increase almost linearly, the recovery time of FCFS-IR is much longer than that of FCFS. For example, the recovery time of FCFS is only 406 ms while that of FCFS-IR is close to 13 seconds when the log size is 5 GB. Because FCFS's recovery mechanism relieves the burden of NVMM write operations caused by checkpointing, this recovery performance improvement with FCFS is due to the amortized checkpointing overhead. Therefore, FCFS provides high system availability.

F. Real Application Performance

In this section, we present the evaluation results for three real-world applications, including SQLite [4], MySQL [5], and Kyoto Cabinet [6], respectively, to see how many performance benefits they can obtain from FCFS. We compare the performance and the NVMM write size of the FCFS-based applications, the original versions, and the no-consistency (NC) versions. The results are shown in Figure 12 and Figure 13 respectively. The original version represents the unmodified application which guarantees data consistency. In contrast, the NC version turns off the transactional part of each application which provides no consistency guarantees. Both the original and the NC versions run on an NVMM-optimized file system where we choose PMFS [10] by default. The performance of the NC version indicates the corresponding theoretical lower bound for providing data consistency in each application.

1) SQLite: We use the mobibench [39] benchmark tool to evaluate the SQLite performance. This benchmark randomly performs 10,000 database row update operations with a single thread. The value size is randomly selected from 0 to 16 KB. The SQLite is configured to use 4 KB page size by default. Moreover, the original SQLite runs in two modes, including the roll-back journal (RBJ) mode and the write-ahead logging (WAL) mode, respectively, while the NC version runs in the journal off mode.

We observe that the FCFS-based SQLite outperforms the original SQLite with the RBJ and WAL mode by 80% and 46% respectively. This performance gain is due to FCFS's efficient failure-consistent update protocol. The original SQLite with the WAL mode runs checkpointing until the WAL becomes about 1,000 pages in size [33]. Therefore, the NVMM write size of the WAL method is 24% lower than that of the RBJ approach because some writes can be combined during checkpointing. However, the WAL-based SQLite still generates 56% more writes than the FCFS-based one due to the constant checkpointing mechanism. Moreover, the checkpointing of the WAL-based SQLite will be run automatically by the foreground process by default, which also impacts the system performance. On the contrary, FCFS performs checkpointing asynchronously and concurrently, thereby getting better performance than both the RBJ and WAL approaches.

2) MySQL: We use two popular database benchmark tools, including Sysbench [37] and YCSB [40], to evaluate the performance of the MySQL database. We set Sysbench to the OLTP mode and run the benchmark on a MySQL database table that has 10,000,000 rows with 16 client threads for 10 minutes. YCSB provides six workloads to imitate web applications' data access models. We only evaluate workload A using the MySQL implementation of the YCSB benchmark, because the rest workloads are read-intensive where the FCFS-based MySQL yields performance similar to the original

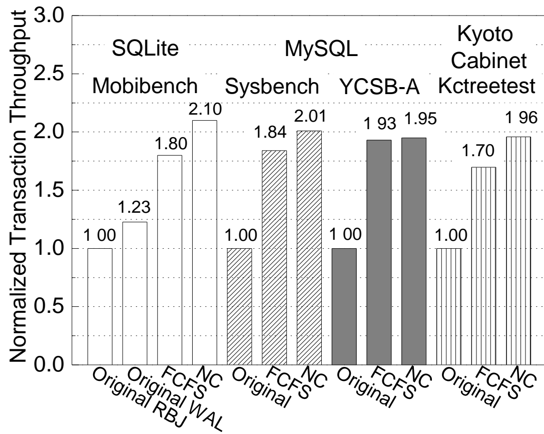


Fig. 12. Performance Comparison for Real-Application Evaluations. *Normalized to the performance of the original application.*

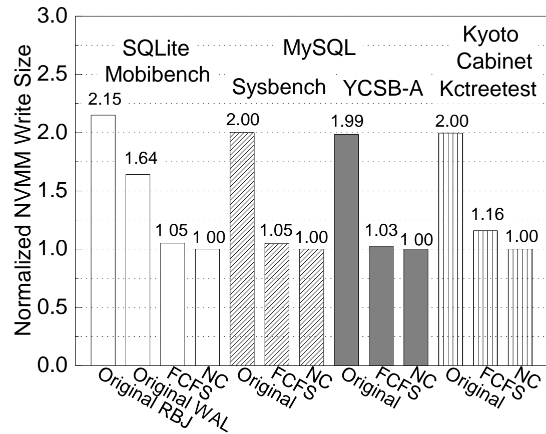


Fig. 13. NVMM Write Size Comparison for Real-Application Evaluations. *Normalized to the NVMM write size of the NC system.*

version. The NC-based MySQL server is started with the `innodb_doublewrite` option being set to the off mode.

We observe that the FCFS-based version shows 84% and 93% better performance than the original version for the Sysbench and YCSB benchmark respectively. This performance gain is attributed to two main reasons: (1) we find that the default InnoDB page size is 16 KB, implying that the I/O size of most database operations is no less than 16 KB, which makes the storage overhead becomes the main system performance bottleneck as Section VI-D1 discussed, and (2) the large reduction of the NVMM write size of the FCFS-based protocol compared to the original InnoDB's double-write mechanism significantly reduces the storage overhead, thereby leading to improved performance.

3) Kyoto Cabinet: We use Kyoto Cabinet's *kctreetest* [6] utility to evaluate the performance of Kyoto Cabinet. We run *kctreetest* with 6 concurrent threads and each thread randomly inserts 10,000 key-value pairs, with each pair containing 0-16 KB of randomly generated data, and then reads the keys 10,000 times. In addition, the database is opened with the auto transaction option to ensure data consistency. The NC-based Kyoto Cabinet is implemented by manually annotating the journal area in the original source code.

As shown in the figures, the original Kyoto Cabinet generates about 2× more writes than the NC-based version because it uses the write-ahead logging technique to support data consistency. As a comparison, the FCFS-based version only has about 1.16× more writes than the NC-based version which is attributed to the effects of the *Selective Checkpointing Algorithm*. The large reduction of the NVMM write size makes the performance of the FCFS-based version 70% higher than the original one.

VII. Related Work

File System Consistency. In the past several decades, file systems have devised a variety of different techniques to prevent their data structures from corruption. Journaling [41], [10], [42], [43], [44], copy-on-write [11], [45], [46], [47], and soft updates [48] can handle inconsistency upon system failures. In addition, recent study NOVA [13] also employs a log-structured file system instead of journaling to provide strong consistency guarantees on NVMM storage. However, NOVA uses block-based (i.e., 4 KB) copy-on-write for each file update, which would incur large write amplification for small write operations because it needs to copy the unmodified data from the old block to the new block. By contrast, while FCFS also writes the latest data to a new block (i.e., pending block) at the logging phase, it only needs to perform the copy-on-write at the cacheline granularity. Therefore, FCFS's transaction mechanism can work well with both small and large transactional updates.

More importantly, none of the above-mentioned file systems provides transactional primitives for applications to perform multiple file operations selectively and atomically (see Table I), implying that they can only guarantee the file system level consistency rather than the application-level consistency. For this reason, applications use their own ad-hoc update protocols (e.g., journaling) to achieve failure consistency when running on these file systems. Unfortunately, these application-specific update protocols are complex and error-prone because they are unaware of the persistence properties of the underlying file systems [14], [15]. Moreover, because these file systems can not eliminate the journaling overheads of the applications, they also hurt application's performance. Therefore, FCFS is the first work to provide the application-level consistency with high performance on NVMM storage.

As PMFS is not going to be maintained publicly [49]. We believe our proposed mechanisms can also work on top of NOVA which is sort of PMFS's subsidiary. To this end, we can alter NOVA's block-based copy-on-write scheme with a cacheline-level one, and then add our proposed volatile index (i.e., the *TLVI* technique in Section III-D) to track the fine-grained file data and the proposed checkpointing technique (i.e., the *CSC* technique in Section III-E) to reduce the overhead of checkpointing. Moreover, FCFS's transactional interfaces are also needed to be integrated to support the application-level transactions.

Transactional File Systems. To support application's transaction at the file system level, a series of transactional file

TABLE I
COMPARISON OF DIFFERENT FILE SYSTEMS ON NVMM STORAGE.

Property	Ext4-writeback	Ext4-ordered	Ext4-datajournal	BPFS [11]	PMFS [10]	NOVA [13]	Valor [25]	FCFS
Atomicity								
Single Metadata Operation	Y	Y	Y	Y	Y	Y	Y	Y
Single Data Operation	N	N	Y	Y	N	Y	Y	Y
Selective Multiple Operations	N	N	N	N	N	N	Y	Y
Consistency Level	F	F	F	F	F	F	A	A
Performance	L	L	L	H	H	H	L	H

Note: To better distinguish the consistency guarantees among traditional block-based file systems, we assume that NVMM provides sector-level atomicity. Legend: Y - Yes, N - No; F - File System (i.e., file system level consistency), A - Application (i.e., application-level consistency); L - Low, H- High. We observe that only FCFS provides the application-level consistency with high performance on NVMM storage.

systems [24], [25], [26], [27], [28], [21] have been proposed to enable transactional file accesses. These transactional file systems can be partitioned into two categories: (1) transactional file systems providing full ACID properties [24], [25], [26], [27] implement both failure consistency and concurrency control but support only strong isolation, which prevent applications from optimizing their performance by the relaxation of isolation; and (2) transactional file systems providing only AD properties [28], [21], however, only guarantee failure consistency, thereby giving application developers more freedom to optimize the application performance according to different isolation semantics.

FCFS is motivated by the second category but is designed and optimized for NVMM storage rather than block-based storage devices (e.g., hard disk or NAND flash). In contrast, all existing transactional file systems are designed and optimized for block-based storage devices, which highly rely on the careful management of the in-memory structures in the OS page cache. Therefore, they will introduce the high copy and software stack overheads between the OS page cache and NVMM storage, leading to disappointing performance on NVMM storage [10], [8]. Moreover, Valor [25] resorts to the costly disk-optimized logging technique to support transactions which further introduces redundant data copies. CFS [21] eliminates the logging overhead but relies on the transactional flash storage. [28] also requires the file system to support per-file writable snapshots. Therefore, none of them is applicable to NVMM storage because of the poor performance of the page-cache based design on NVMM storage and the additional hardware or software requirements.

To overcome this issue, FCFS is designed to address the challenges of how to provide fast transactional file accesses on NVMM storage by leveraging NVMM's unique characteristics of byte addressability and high concurrency but without relying on the page-cache layer. Note that FCFS's design does not introduce the page cache and generic block layer overheads, because both logging and checkpointing operations are directly performed to NVMM storage via the memory interface.

Transactional Programming Models for NVMM Storage. To eliminate the kernel and file abstraction overheads in order to enable fast user-mode access to NVMM storage, there have been a lot of efforts to provide new transactional models or interfaces for programming with NVMM storage [16], [22], [23], [17]. In such cases, while applications can fully exploit the NVMM performance, they also lose the important file system features such as sharing semantics and global naming [9]. Different from them, FCFS provides transactional accesses based on file I/O interfaces so that applications can be ported to it more easier, because most existing applications [3], [4], [5], [6], [7] are built and implemented using traditional file I/O interfaces.

Hardware-based Transaction Mechanisms for NVMM Storage. To overcome inefficiencies associated with logging and shadow paging, recent research [50], [51] has proposed to extend the CPU hardware to guarantee failure consistency efficiently on NVMM storage. For example, Kiln [50] employs a non-volatile last-level CPU cache to enable atomic in-place updates. However, this design cannot efficiently accommodate large-granularity persistent updates in database and file system applications due to the limited capacity of the non-volatile cache [52]. By contrast, FCFS's transaction mechanism can support large transactions of even several gigabytes which is more suitable for file system applications. ThyNVM [51] also uses a hardware-assisted checkpointing approach to support software-transparent failure consistency. While Kiln and ThyNVM effectively improve the transaction performance on NVMM storage, both of them require hardware modifications inside CPUs. Different from them, FCFS reduces the overhead towards supporting failure consistency using a software-based approach which does not require hardware support.

VIII. CONCLUSION

FCFS is the first NVMM-optimized file system which enables both correctness and high performance for applications to consistently update their data on NVMM storage. To enforce failure consistency efficiently on NVMM storage, FCFS provides a series of easy-to-use file-based interfaces and employs an NVMM-optimized WAL scheme to reduce the overhead towards supporting failure consistency by fully leveraging NVMM's byte addressability and high concurrency but without relying on the page-cache layer. Our experiments demonstrate that FCFS's failure-consistent update protocol and FCFS-based applications significantly outperform conventional protocols and original applications, respectively.

ACKNOWLEDGMENT

We thank our shepherd Jishen Zhao and the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the National High Technology Research and Development Program of China (Grant No. 2013AA013201). Jiwu Shu is the corresponding author.

REFERENCES

[1] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST '15)*, May 2015, pp. 1–10.

[2] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, 2015, pp. 3–18.

[3] "Vim the editor," http://www.vim.org/index.php.

[4] "Sqlite transactional sql database engine," http://www.sqlite.org/.

[5] "Mariadb: an enhanced, drop-in replacement for mysql," https://mariadb.org/.

[6] "Kyoto cabinet: a straightforward implementation of dbm," http://fallabs.com/kyotocabinet/.

[7] "Google's leveldb key-value store," https://github.com/google/leveldb.

[8] J. Corbet, "Supporting filesystems in persistent memory," https://lwn.net/Articles/610174/, 2014.

[9] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, 2014, pp. 14:1–14:14.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, 2014, pp. 15:1–15:15.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, 2009, pp. 133–146.

[12] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, 2011, pp. 39:1–39:11.

[13] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, Feb. 2016, pp. 323–338.

[14] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014, pp. 433–448.

[15] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014, pp. 449–464.

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, 2011, pp. 91–104.

[17] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, 2011, pp. 105–118.

[18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[19] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013, pp. 197–212.

[20] K. Suzuki and S. Swanson, "The non-volatile memory technology database (nvmdb)," Department of Computer Science & Engineering, University of California, San Diego, Tech. Rep. CS2015-1011, May 2015.

[21] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight application-level crash consistency on transactional flash storage," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, Santa Clara, CA, Jul. 2015, pp. 221–234.

[22] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST '15)*, May 2015, pp. 1–13.

[23] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*, Oct 2014, pp. 216–223.

[24] "Transactional ntfs (txf)," https://msdn.microsoft.com/zh-cn/library/windows/desktop/bb968806(v=vs.85).aspx.

[25] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling transactional file access via lightweight kernel extensions," in *Proccedings of the 7th Conference on File and Storage Technologies (FAST '09)*, 2009, pp. 29–42.

[26] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter, "Improving server applications with system transactions," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, 2012, pp. 15–28.

[27] M. Seltzer *et al.*, "Transaction support in a log-structured file system," in *Proceedings of the Ninth International Conference on Data Engineering*. IEEE, 1993, pp. 503–510.

[28] R. Verma, A. A. Mendez, S. Park, S. S. Mannarswamy, T. P. Kelly, and C. B. M. III, "Failure-atomic updates of application data in a linux file system," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, Feb. 2015, pp. 203–211.

[29] S. Kannan, A. Gavrilovska, and K. Schwan, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, Feb 2014, pp. 512–523.

[30] I. Cooperation, "Intel architecture instruction set extensions programming reference," https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[31] "Trees i: Radix trees," https://lwn.net/Articles/175432/.

[32] "Atomic commit in sqlite," https://www.sqlite.org/atomiccommit.html.

[33] "Write-ahead logging," https://www.sqlite.org/wal.html.

[34] "The innodb storage engine," https://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.

[35] "Innodb disk i/o," http://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html.

[36] "Fundamental specifications of kyoto cabinet version 1," http://fallabs.com/kyotocabinet/spex.html.

[37] "Sysbench benchmark tool," https://launchpad.net/sysbench.

[38] "mysqlcheck - a table maintenance program," http://dev.mysql.com/doc/refman/5.7/en/mysqlcheck.html.

[39] E. LABORATORY, "Mobibench benchmark tool," http://www.mobibench.co.kr/.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, ser. SoCC '10, 2010, pp. 143–154.

[41] "Journaled file system technology for linux," http://jfs.sourceforge.net/.

[42] S. C. Tweedie, "Journaling the linux ext2fs filesystem," in *Proceedings of the Fourth Annual Linux Expo*, 1998.

[43] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1349–1360, May 2015.

[44] M. Cao, S. Bhattacharya, and T. Tso, "Ext4: The next generation of ext2/3 filesystem." in *Linux Storage and Filesystem Workshop (LSF)*, 2007.

[45] "Btrfs," https://btrfs.wiki.kernel.org/index.php/Main_Page.

[46] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[47] "Zfs," https://en.wikipedia.org/wiki/ZFS.

[48] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: A solution to the metadata update problem in file systems," *ACM Trans. Comput. Syst.*, vol. 18, no. 2, pp. 127–153, May 2000.

[49] "Persistent memory file system," https://github.com/linux-pmfs/pmfs.

[50] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, 2013, pp. 421–432.

[51] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, 2015, pp. 672–685.

[52] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, 2014, pp. 153–165.