IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 20, 2019, accepted November 8, 2019, date of publication December 23, 2019, date of current version January 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2960838

An Empirical Performance Evaluation of Transactional Solid-State Drives

YONGSEOK SON[1], HEON YOUNG YEOM[2], AND HYUCK HAN[3]
[1]School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea
[2]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[3]Department of Computer Science, Dongduk Women's University, Seoul 02748, South Korea

Corresponding author: Hyuck Han (hhyuck96@dongduk.ac.kr)

This work was supported in part by the Chung-Ang University Research Grant, in 2018, and in part by the National Research Foundation of Korea (NRF) under Grant 2015M3C4A7065645.

ABSTRACT Solid-state drives (SSDs) have accelerated the architectural evolution of storage systems with several characteristics (e.g., out-of-place update) compared with hard disk drives (HDD). Out-of-place update of SSDs naturally can support transaction mechanism which is commonly used in systems to provide crash consistency. Thus, transactional functionality has been recently implemented inside solid-state drives (SSDs). However, this approach must be re-evaluated for enterprise storage with a standard interface to investigate their benefits in a more realistic and standard fashion. In this article, we explore the implications and challenges of transactional SSDs with different experiments. To evaluate the potential benefit of transactional SSDs, we design and implement the transactional functionality in a Samsung enterprise-class and SATA-based SSD (i.e., SM843TN) called TxSSD. We modify the local file systems (i.e., ext4 and btrfs) and a distributed parallel file system (i.e., Lustre) to utilize TxSSDs. Our modified file systems with TxSSDs provide crash consistency without redundant writes. We evaluate our file systems by using multiple micro and macro benchmarks. We analyze the performance results and demonstrate that TxSSDs may generate an overhead for supporting transactional functionality inside SSD.

INDEX TERMS Solid-state drives, file system, distributed file system, performance, consistency, transaction.

I. INTRODUCTION

Flash memory is widely used for storage devices from single to large-scale high performance systems since it provides lower latency, lower power consumption, and higher throughput than hard disk drives (HDDs) [5], [43]. In addition to the advantages, as the cost per byte is falling while the storage capacity is increasing, large-capacity flash memory devices are more commonly employed for high-end desktops and enterprise storage servers. Large-scale datacenters host many simultaneously running applications, cater to many millions of active users, and service billions of transactions daily. Thus, the load on the storage systems in these datacenters has been enormously increasing. Flash-based solid-state drives (SSDs) are attractive solutions to meet these performance demands for large-scale datacenters [8], [21].

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

To match the required load, the storage systems must be designed to scale in performance, and clustering techniques are used to provide scalability. For example, distributed parallel file systems assemble the cluster elements into one large and seamless storage system. The file system ensures that all clients have a consistent view of the file system by handling the locations of the files and transmission of their data. The file system distributes blocks in a file to different storage locations by using a network protocol to provide high scalability. This approach enhances the scalability of the storage systems and provides higher performance to clients. However, scalability and performance can be negatively affected by providing crash consistency to ensure that client data is recovered consistently from a system crash. Thus, the file systems have considered the trade-off between performance and crash consistency to provide better-quality service. Generally, a higher level of consistency makes the file systems more consistent but negatively affects the performance and endurance of flash-based SSDs.

Most distributed parallel file systems, including Lustre [38], Ceph [46], Gluster [10], and HDFS [4], [34], [40] rely on local file systems to support crash consistency. Most local file systems provide crash consistency to applications by using journaling or copy-on-write (CoW) techniques. Journaling file systems, such as ext4 [26], XFS [44], ReiserFS [13], JFS [15], and NTFS [9], provide transaction processing for atomicity and durability by using write-ahead logging (WAL) [12]. The file systems write data and metadata to the journal area before writing them to the original area. When hardware or software failures occur, the file systems recover the data and metadata by replaying the written journals. CoW file systems (e.g., btrfs [36], LFS [37], and ZFS [7]) also provide transaction processing by using out-of-place update techniques. When data and metadata are written, CoW file systems copy, modify, and flush them when a transaction commits. They leave older versions of the data and metadata in a storage medium until garbage collection is executed to remove them. However, although these techniques provide crash consistency, they reduce the I/O performance since the data and metadata are written twice. It is a challenge to escape the trade-off between crash consistency and performance.

To provide crash consistency without sacrificing performance, previous studies [18], [30], [33] support transaction functionality inside an SSD by using its characteristic (i.e., out-of-place update). They offload the burden of guaranteeing the transactional atomicity from a host system to flash-based SSDs. Since flash-based SSDs do not allow any page to be overwritten in place, a page update leaves the existing page intact and writes the new content into a clean page at another location [18]. This copy-on-write strategy is adopted by most flash-based SSDs. Since the flash-based SSDs internally perform out-of-place updates, transaction processing inside SSDs can alleviate the trade-off between performance and consistency. This article is in line with these previous studies [18], [30], [33] in terms of the study of transactional SSDs. In contrast, we perform extensive performance studies for a transactional SSD which is both enterprise-class and SATA-based for a more realistic and standard fashion.

In this article, we introduce TxSSD, a transactional SSD with an enterprise-class and SATA interface, and evaluate it with diverse file systems and workloads. TxSSD supports transactional functionality inside SSD by using the nature of flash memory. To do this, we modify the flash translation layer (FTL) of TxSSD and explain how we design and implement transactional functionality in a Samsung SM843TN SSD, which is widely used in datacenters due to the low latency and high throughput. To make the file systems exploit TxSSD, we modify the existing local file systems (i.e, ext4 and btrfs) and evaluate the file systems by using file I/O and online transaction processing (OLTP) workloads. Furthermore, we modify a distributed parallel file system (i.e., Lustre) and evaluate the file system by using HPC workloads in a cluster system.

TxSSD-aware file systems preserve crash consistency and retain transaction models of the existing file systems without redundant writes for metadata and data. In TxSSD-aware file systems, legacy and new applications benefit from file systems transparently without any modification. With the experimental results, we analyze the performance results and demonstrate that TxSSD can generate an overhead due to support the transactional functionality. We also disclose and analyze the reason for the overhead. In our previous work [41], we focused on the study of local file systems in a single TxSSD. This article extends our scheme to distributed file systems in a cluster environment. To the best of our knowledge, this is the first study that provides a performance evaluation for a distributed parallel file system as well as local file systems on transactional SSDs at enterprise level with a SATA-based interface.

The main contributions from this study are as follows:

- We design and implement the transactional functionality in an enterprise-class and SATA-based SSD called TxSSD.
- We modify local file systems and a distributed parallel file system to utilize the benefit of TxSSD.
- We show an empirical evaluation from our comprehensive performance study using diverse workloads.
- We show that transaction support in enterprise-class and SATA-based SSDs can incur an overhead and analyze the reason for the overhead.

The rest of this article is organized as follows. Section II describes the background and motivation. Section III explains the design and implementation. Section IV shows the experimental results. Section V discusses related work. Section VI discusses a summary and implications of our study. Finally, Section VII concludes this article.

II. BACKGROUND AND MOTIVATION

A. CRASH CONSISTENCY IN LOCAL FILE SYSTEMS

Local file systems such as ext4 [26], xfs [44], f2fs [20], btrfs [36], etc provide crash consistency by using a variant of write-ahead logging [12] or copy-on-write schemes. In the midst of many file systems, ext4 [26] is the most widely used file system in Linux and more general than other file systems. Ext4 uses a fork of the journaling block device (JBD) called JBD2 which is a variant of write-ahead logging [12].

Ext4 supports three journaling modes such as writeback, ordered, and data journaling mode [16], [45]. Each mode has a different consistency level and different performance according to the level. The writeback mode supports transaction processing only for metadata writes. This mode does not keep the write order between the metadata and data. The ordered mode (default) supports transaction processing only for the metadata writes but supports stronger consistency compared with the writeback mode by keeping the write order between the metadata and data. Before the metadata is written to the journal location, data is written to their original location. The data journaling mode performs transaction

processing for both metadata and data. It writes both metadata and data to the journal location before they are written to the original location. This mode supports the strongest consistency with data integrity (crash consistency). However, it shows the lowest performance because of redundant data writes.

Btrfs [36] is a CoW file system. It allows atomic transactions without a separate journal. Btrfs maintains B-trees for both data and metadata and supports two modes such as datacow and nodatacow [36]. The datacow mode (default) performs out-of-place updates for the data and metadata by creating a new version of an extent or a page at a different location, which prevents a partial update on power failures. This mode performs garbage collection for both data and metadata. Meanwhile, the nodatacow mode performs only out-of-place updates and garbage collection for metadata. This supports higher performance compared with the default mode. But, it does not provide crash consistency due to the lack of data integrity.

B. CRASH CONSISTENCY IN DISTRIBUTED PARALLEL FILE SYSTEMS

Distributed parallel file systems, including Ceph, Gluster, and Lustre, distribute file data across multiple servers and support concurrent access by multiple tasks of a parallel application. Among the many distributed parallel file systems, Lustre is generally used for large-scale cluster computing, such as scientific supercomputing and industry. It can be part of clusters with tens of thousands of client nodes, tens of petabytes of storage on hundreds of servers, and a terabyte per second of aggregate I/O throughput. This makes Lustre a popular choice for large datacenters. Thus, the Lustre file system has attracted increasing attention from research and industry communities.

Traditionally, a Lustre file system has three major functional units which are the metadata server (MDS), the object storage server (OSS), and the client. The Lustre file system includes one or more MDSs that have one or more metadata targets (MDTs) that store namespace metadata, such as filenames, directories, access permissions, and file layout. Also, a MDT has a dedicated file system that controls file access and notifies clients the layout of the objects that make up each file. The OSS provides file I/O service and network request handling for one or more object storage targets (OSTs) that store file data. The number of objects per file is configurable by the user and can be tuned to optimize the performance for a given workload. An OST has a dedicated file system that exports an interface for read/write operations. OSTs and MDTs use a variant of ext4 called ldiskfs, which has the same journaling mechanism as ext4.

Lustre presents all Lustre clients with a unified namespace for all of the files and data in the file system using standard POSIX semantics. Lustre allows concurrent and coherent read and write access to the files. When a client tries to read from or write to a file, it performs a filename lookup on the MDS and fetches the file layout from the MDT object for the file. The file layout is stored in a MDT identified by the file identifier (FID), which contains information about where the file data is located on the OST(s). The client then uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects are stored.

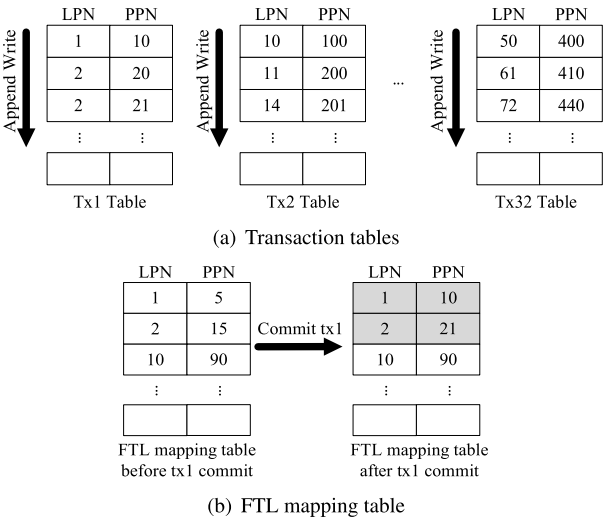


FIGURE 1. Transaction (tx) and FTL mapping tables (logical page number (LPN), physical page number (PPN)).

III. DESIGN AND IMPLEMENTATION

A. TRANSACTIONAL SOLID-STATE DRIVE

In this section, we explain the design and implementation for a transactional functionality inside SSDs (called TxSSD). We exploit the out-of-place updates of SSDs while considering the features of enterprise and SATA based SSDs. Also, we use the buffer memory with supercapacitors inside the device and choose more efficient data structures for SSD. To support the transactional functionality, TxSSD leverages multiple table-based data structures to store information of transaction in the FTL. TxSSD uses a table per transaction to avoid the overhead from accessing the mapping information of all transactions in a table. Figure 1 depicts FTL mapping tables and transactions inside TxSSD. The FTL in TxSSD has 32 transaction tables (called tx tables) per transaction ID as shown in Figure 1(a). The maximum number of concurrent transactions is limited to 32 due to the restricted size of the memory inside SSD. Each table supports 60,000 entries and each entry is 20 bytes for a logical page number (LPN) and a physical page number (PPN). The required space is 36.6 MiB for the tx tables. The number of tx tables (32) are sufficient to process the transactions of file systems. It is because most file systems such as ext4, btrfs, xfs leverage a single compound transaction scheme which provides two concurrent transactions such as a running transaction and a committing transaction at most [32], [45]. Thus, under existing file systems, we choose the enough number of tx tables and txIDs and so that the transaction overflow never occurs. If a file system supports multiple transactions, TxSSD supports 32 transactions concurrently now. If the number of

TABLE 1. Transactional operations supported by TxSSD.

Operation Name	Description
Write(LBA, length, txID...)	Writes data and stores mapping information in the tx table associated with the given txID
Read(LBA, length, txID...)	Reads data by referencing the mapping information stored in the tx table associated with the given txID
Commit(txID)	Remaps the entries from the tx table associated with the given txID to those to the FTL mapping table
Abort(txID)	Discards mapping information in the tx table associated with the given txID
Recovery	Discards uncommitted mapping information in all transaction tables

transaction is larger than the 32 transactions, we can also increase the number of tx tables and TxIDs. In addition, a file system on top of our TxSSD have to commit a transaction before the number of operations reaches the limit because the number of operations per transaction is restricted to 60,000.

We design and implement TxSSD based on SM843TN SSD. To enable transactional functionality, we do not include new SATA commands and instead employ the reserved count field (6 bits) and the NCQ tag field (5 bits) in the existing commands. Table 1 depicts transactional operations supported by TxSSD. We use the first 5 bits of the reserved count field for the txID (called txID field) and the last bit for the commit flag (called commit field). For the transactional read and write operations, we store the transaction ID of the file system to the txID field in the read and write commands of SATA. In case of the commit operation, we use both the txID field and the commit field in the write command of SATA. We extend the flush command of SATA and assign 1 and 2 to the NCQ tag field[1] for abort and recovery operations, respectively, and set the txID field of the flush command.

When a write operation is performed with a new txID, TxSSD stores the mapping information between the LPN and PPN in the tx table. However, TxSSD delays propagating the mapping information to the FTL table. If a page requested by the write operation already exists, TxSSD does not perform garbage collection for the existing page until the commit request arrives. When updating the transaction entries, instead of in-place update, we choose an append operation. It is because the in-place update can incur an overhead due to the memory access time searching for appropriate entries.[2] Thus, if there is more than one update for the identical LPN, TxSSD just appends the mapping information at the end of the tx table instead of searching and updating the previous entry as shown in an example of the tx1 table in Figure 1(a). In term of read operation, when TxSSD receives a read request with a txID, it finds and returns the page according to the corresponding LPN in the tx table. If there are several entries for the LPN, TxSSD returns the most recently updated page for consistency. If a requested LPN cannot be found in the tx table, TxSSD returns the page in the FTL mapping table.

As shown in Figure 1(b), TxSSD propagates the entries in the tx table to the FTL mapping table after receiving a commit request. TxSSD performs garbage collection for the old pages associated with the tx table after the commit operation is completed. Unlike previous approaches [18], [27], in our scheme, all entries in the tx table can be remapped without checking whether the entries are active or committed since TxSSD includes each tx table per transaction. During the commit procedure, if there are multiple entries for identical LPNs in the tx table, the mapping information of the last written entry is applied for remapping.

When a power outage occurs during the commit procedure, all information is preserved using the supercapacitor. TxSSD applies the mapping information in the tx table to the FTL mapping table when the commit request is received at TxSSD. In abort operation, TxSSD discards all entries in the given tx table, enabling garbage collection for pages associated with the transaction. The recovery operation discards all uncommitted mapping information in all tx tables after a power outage.

B. ENHANCED EXT4 (E2XT4)

Our enhanced ext4 (e2xt4) file system is based on transaction model (i.e., single compound transaction) and ordered mode of existing ext4. However, e2xt4 does not perform redundant metadata writes to SSD. For metadata, e2xt4 redirects the journaled metadata to its original location and disables the checkpoint. When a transaction commits, e2xt4 performs the transactional write operations for the data and metadata. To do this, when the data and metadata are transferred to storage, e2xt4 obtains the current transaction ID from the existing journaling scheme, remaps the transaction ID to an unused txID of TxSSD, and transfers the transaction ID through I/O descriptions such as BIO structure. Then, e2xt4 issues a flush command,[3] however, TxSSD may not flush all pages from the device buffer to the flash memory[4] due to the supercapacitors. Finally, e2xt4 creates a commit block and writes it to TxSSD.

When ex2xt4 performs write operation for the commit block, it writes the commit block with the txID and a commit flag to TxSSD (a transactional write operation). When TxSSD receives the commit block, it applies the mapping information in the tx table to the FTL mapping table. TxSSD writes all pages to the flash memory according to its policy. After the commit operation, TxSSD empties the tx table and allows garbage collection for the old pages.

[1]The NCQ tag field is not used for the flush operation of the file system.

[2]The memory access time for 4 bytes inside SSD is single-digit microseconds.

[3]Most commodity SSDs store in-processing pages to flash memory to process the flush command, which leads to a significant overhead.

[4]When the flush command reaches TxSSD, TxSSD flushes pages in the device buffer to the flash memory only when the number of dirty pages exceeds the threshold. The size of device buffer is about 70 MiB, which can be protected by the supercapacitor.

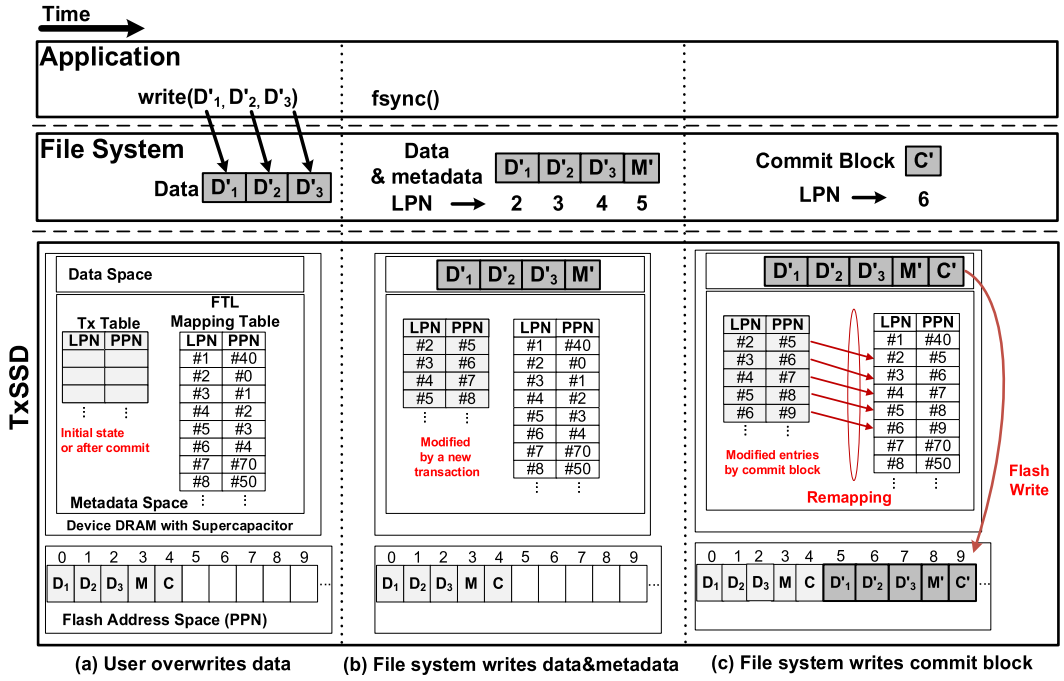


FIGURE 2. Enhanced ext4 file system (e2xt4) with TxSSDs.

Before the commit operation, if the written data is not in the page cache or a direct I/O is performed, transactional read operations are performed. E2xt4 reads the data by referencing the tx table because the mapping information associated with the data exists in the tx table. If an I/O operation is failed, e2xt4 performs an abort operation for the txID.

When a system failure or a power outage occurs, TxSSD emergently stores all information, including the tx tables and FTL mapping table, to extra area of the flash memory called a safe region. If the last entry in the tx table is a commit block, TxSSD remaps all mapping information of the tx table before shutdown. When a recovery operation (i.e., mount procedure) is performed, TxSSD discards all mapping information in the uncommitted tx tables.

Figure 2 shows an example of an `fsync` call and the commit procedure after `write` calls in e2xt4 on top of TxSSD. As shown in the figure, an application updates the data (D'_1, D'_2, and D'_3 pages), which is written to the page cache in the file system. The LPNs of D'_1, D'_2, and D'_3 are 2, 3, and 4, respectively. E2xt4 also updates the metadata page (M') with the LPN 5. The old pages for D'_1, D'_2, D'_3, M', and commit block (C') are stored in TxSSD. From the FTL mapping table in Figure 2(a), we can see that the PPNs of the old pages for D'_1, D'_2, D'_3, M', and C' are 0, 1, 2, 3, and 4, respectively. If a running transaction exists, the write operations are merged into the current transaction; otherwise, e2xt4 starts a new transaction. After the updates, the application calls `fsync` to permanently store the data and metadata by committing the transaction. When the write operations are performed by a page flusher or the `fsync` call, e2xt4 assigns the updated data and metadata pages in the page cache to a transaction with a newly created txID or compounds them with an existing transaction with an existing txID.

Then, e2xt4 writes the data and metadata pages (D'_1, D'_2, D'_3, and M' pages) with the transactional write operation. When TxSSD receives the pages, it makes LPN/PPN mappings for the pages in the tx table of the device. From the tx table of Figure 2(b), the PPNs for the D'_1, D'_2, D'_3, and M' pages are 5, 6, 7, and 8, respectively. Then, e2xt4 issues a flush command and creates a commit block (the C' page in Figure 2(c)) and writes it to TxSSD. To write the commit block, e2xt4 performs a transactional write operation with the txID and the commit flag to TxSSD. When TxSSD receives the page for the commit block (PPN for the C' page is 9 now), TxSSD propagates the LPN/PPN mapping information in the tx table to the FTL mapping table. Later, TxSSD stores all pages ($D'1$, $D'2$, $D'3$, M', and C' pages) to the flash memory according to the write policy of TxSSD. After then, TxSSD empties the tx table and permits garbage collection for old pages. From the description above, we see that the write traffic of e2xt4 is approximately the same as that of ext4 without journaling, but e2xt4 provides strong consistency with data integrity as the data journaling mode does (crash consistency).

C. ENHANCED BTRFS (EBTRFS)

Enhanced Btrfs (ebtrfs) is on the basis of the nodatacow mode and transaction model of btrfs. In contrast, ebtrfs does not perform copy-on-write for metadata like the data operations of the nodatacow mode. Ebtrfs writes both metadata and data into their original locations without garbage collection. The commit procedure in ebtrfs is similar to that of e2xt4.

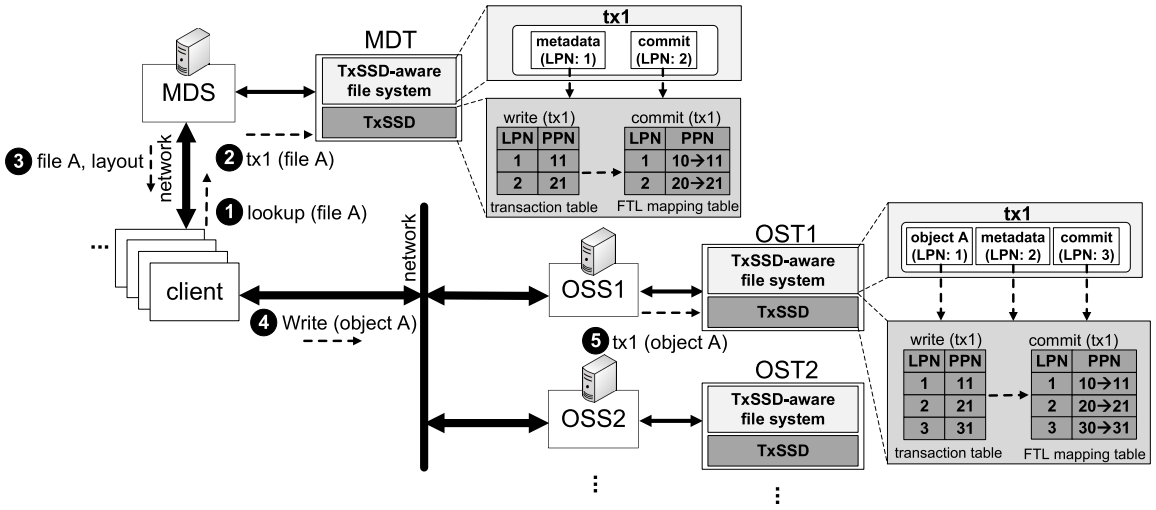


FIGURE 3. Enhanced Lustre file system (elustre) with TxSSDs.

Thus, ebtrfs maintains txIDs and keeps the order between the metadata and data updates. When ebtrfs performs a commit operation, it writes metadata and data to TxSSD as transactional operations. And then, ebtrfs issues a flush command and performs a write operation for the superblock as a commit block with a commit flag. Other operations (e.g., read, abort, recovery operations) of ebtrfs are almost similar to those of e2xt4.

D. ENHANCED LUSTRE (ELUSTRE)

Our elustre is based on the ordered mode and the transaction model of ldiskfs. However, elustre does not perform redundant write operations for metadata to SSD in both MDT and OST. Thus, elustre supports transactions and crash consistency without any redundant writes for both metadata and data like e2xt4 and ebtrfs. The metadata managed by MDT contains the layout about the OSTs on which the file objects are located. This layout information is stored in the extended attribute (xattr) section of the inode in the MDT. Elustre processes the transactions for the metadata stored in the MDT and both metadata and data stored in the OST. The metadata operations in the MDT and the OST are similar to those of e2xt4. We write the metadata in a transaction to its original location and disable the checkpoint operation. When a commit operation for the transaction occurs, the metadata gets associated with the transaction by using the txID and written according to the transaction table.

For the data operations in the OST, we allocate a txID to an object. When a new transaction is generated, the transaction associates the object to be written to storage with the txID. Thus, the objects associated with the transaction are written to storage, and the transaction table related to the txID is updated before their metadata are written to storage as the ordered mode of ext4 does. Meanwhile, the metadata and data are processed as a transaction by using the same txID and a commit block for the transaction.

Figure 3 shows an example of a write operation in elustre. As shown in the figure, each MDT and OST has a txSSD-aware file system and TxSSD. When a client writes an object (object A) for a file, the client looks up the file and creates a transaction (tx1) in the MDT. The txSSD-aware file system in the MDT allocates a txID (e.g., txID: 1) and associates the txID with the modified metadata of the file. Then, the MDT returns the layout of the file to the client. The client gets the layout information of the file, requests the write operation to OST1, and makes a transaction in OST1. The txSSD-aware file system in the OST allocates a txID (e.g., txID: 1) and associates the txID with the modified metadata and data. The modified metadata in the MDT and the modified metadata and data in the OST are flushed to the transactional SSDs with the commit blocks when the transactions are committed.

As shown in the figure, the modified metadata and the commit block in the MDT are included in a transaction; when the two blocks are written to TxSSD, the LPNs/PPNs of the two blocks are mapped to 1/11 and 2/21, respectively, and written in the transaction table. When the transaction commits, the FTL mapping table is modified. This procedure is also performed in the OST. In the OST, the LPNs/PPNs of the three blocks are mapped to 1/11, 2/21, and 3/31, respectively, and written in the transaction table. Similar to the case of the MDT, the FTL mapping table is modified when the transaction commits. Consequently, our elustre supports the transaction processing for both metadata and data without redundant writes by using the transactional SSDs.

E. IMPLEMENTATION

To enable enhanced file systems and TxSSD, we create a table with 32 entries to map the transaction IDs of the file systems to the txIDs of TxSSD. The existing block I/O subsystem cannot issue transactional operations of TxSSD since the subsystem cannot access to the txID number, commit

flag, and recovery/abort flag. Thus, we add three fields to four descriptors (`struct bio`, `struct request`, `struct scsi_cmd`, and `struct ata_queued_cmd`) in the block I/O subsystem. Additionally, we modify the block I/O subsystem to transfer the added fields from the upper to lower layer of the block I/O subsystem. The modified lines of code in e2xt4, ebtrfs, elustre, and the block I/O subsystem are 193, 202, 214, and 109, respectively. This demonstrates that our scheme requires small modifications, and so our scheme can be easily applied to other file systems.

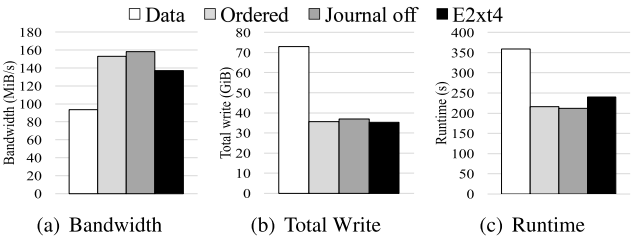


FIGURE 4. FIO results in e2xt4 with 32 threads, 1 GiB file size, 4 KiB request size, and 10 random write operations between fsync.

IV. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETUP

To evaluate local file systems, we use a machine, which has two Intel Xeon CPU E5-2670 (2.6 GHz) (total 16 physical cores), 8 GiB DRAM, SATA 3 interface, and Linux 3.14.3. A separate client machine is used for the OLTP evaluation. The machine has two Intel Xeon CPU E7-8837 (2.67 GHz) with 16 physical cores each (64 cores in total with hyper-threading). For TxSSD, we used SM843TN developed by Samsung. It has a capacity of 240 GiB and is designed for high-performance servers and storage in demanding data-centers by providing a powerful controller and power-loss protection. For comparison, we use unmodified SM843TN SSD with ext4 and btrfs and TxSSD with e2xt4 and ebtrfs. We use the FIO benchmark [14] to measure the file I/O performance and the OLTP benchmark [19] to measure the database performance (i.e., transactions per minute (TPM)).

To evaluate the Lustre file system, we use a cluster system, which consists of 8 identical machines connected by a network. Each machine has an Intel Core CPU i7-4790 (3.60 GHz) with 4 physical cores, which total up to 8 cores with hyperthreading, 32 GiB DRAM, SATA 3 interface, and a 10 GbE network card. We configure one node as a MDS with one MDT, which has a TxSSD and six OSSs with each OST, which has a TxSSD, and one node as a client. All servers run CentOS 7 with a Linux kernel 3.10 patched for Lustre. To measure the performance, we use the IOR benchmark [1] and mdtest [2], which are data-intensive and metadata-intensive workloads, respectively. All experimental results report the average value of ten runs.

B. LOCAL FILE SYSTEM PERFORMANCE

1) FILE I/O PERFORMANCE

We present the performance evaluation of our enhanced local file systems on TxSSD using the FIO benchmark [14]. We configure FIO to perform random write operations using 32 threads. Each thread writes 1 GiB file with 4 KiB request size and different numbers of writes per `fsync` call.

a: ENHANCED EXT4

We compare e2xt4 with the ext4 file system in three different journaling modes such as data journal mode, ordered mode, and journal off mode. We measured the bandwidth, the total amount of writes, and the runtime as shown in Figure 4. Figure 4(a) shows that the bandwidth for data journaling mode is 93.4 MiB/s whereas the bandwidth for the ordered and journal off modes is 153 MiB/s (1.64X) and 158.4 MiB/s (1.70X), respectively. The data journaling mode shows far lower bandwidth compared with the other journaling configurations because there is an overhead of redundant data writes. However, the performance of the ordered mode is similar to that of the journal off mode even though the ordered mode has additional overheads such as redundant metadata writes and flush command. It is because the workload is data-intensive and the SSD with supercapacitor mostly returns to the host without flushing the data in the device cache, resulting in almost no overhead for flushing. E2xt4 shows a higher bandwidth of 137 MiB/s (1.46X) compared to ext4 in the data journaling mode. However, the bandwidth of e2xt4 is 10.5% and 13.5% lower than those of the ordered and journal off modes, respectively. This indicates the presence of transactional support overhead, such as remapping transaction tables in TxSSD.

In terms of the total amount of writes, Figure 4(b) shows that the data journaling mode has written 72.9 GiB. Meanwhile, ordered and journal off modes have written approximately 34.2 GiB. This indicates that the amount of the metadata journal and its overhead is almost negligible compared to the data journal. According to the expectation, e2xt4 also shows almost the same amount of total data writes as the journal off mode. In terms of runtime, Figure 4(c) shows that the data journal mode takes 1.66X and 1.70X more time than the ordered and the journal off modes, respectively. The runtime of e2xt4 is 10.8% longer than that of the ordered mode, which is consistent with the bandwidth results provided in Figure 4(a). These results show that providing crash consistency in the existing ext4 has a significant performance tradeoff in terms of the bandwidth, write amplification, and runtime. Meanwhile, e2xt4 shows relatively small overhead while keeping strong consistency with data integrity.

Figure 6 shows the FIO performance according to different `fsync` parameters. This indicates how many I/O operations to perform before issuing an `fsync` [14]. An `fsync` operation flushes the dirty metadata and data, thus, a higher `fsync` frequency results in a lower performance due to high I/O overhead. As shown in the figure, there are performance drops of 9.4%, 10.1%, 8.2%, and 11.0% as the `fsync` parameter is reduced from 100 to 10 in the data journaling, ordered,

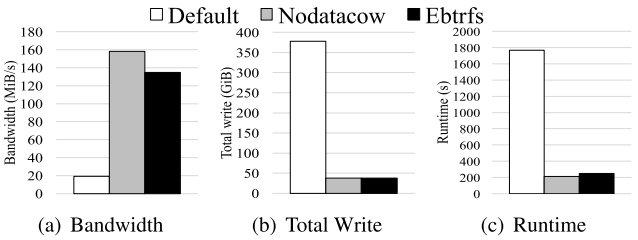


FIGURE 5. FIO results in ebtrfs with 32 threads, 1 GiB file size, 4 KiB request size, and 10 random write operations between fsync.

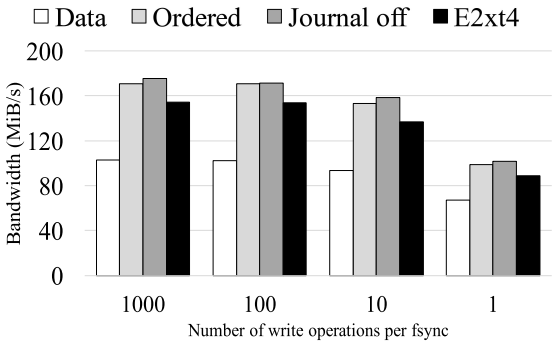


FIGURE 6. FIO results in e2xt4 with different fsync parameters.

journal off modes, and e2xt4, respectively. As the number of write operations per `fsync` is reduced from 10 to 1, we observe the performance reduction of 28.1%, 34.5%, 37.6%, and 35.3%. E2xt4 shows 1.50X, 1.51X, 1.47X, and 1.32X performance of the data journaling mode. By comparing e2xt4 and the ordered mode, the overhead of the transactional support in TxSSD is shown as 9.7%, 9.4%, 10.5%, and 10.3% in the case of 1000, 100, 10, and 1 write operations per `fsync`, respectively.

b: ENHANCED BTRFS

Figure 5 shows the performance of ebtrfs and btrfs using FIO performance. We evaluate the btrfs with the datacow and nodatacow modes. As shown in Figure 5(a), ebtrfs and btrfs with the nodatacow mode improve 7.1X and 8.32X random write performance compared with the default configuration of btrfs. This result shows that btrfs generates a significant performance overhead in trade with crash consistency. Figure 5(b) shows the total amount of written data and runtime. As shown in the figure, there is a larger gap between the btrfs with datacow and nodatacow/ebtrfs It is because datacow not only copies data but also performs garbage collection on obsolete pages by performing discard operations. Figure 7 shows the performance of ebtrfs and btrfs under different `fsync` parameters. Similar to the case of ext4, as the number of write operations per `fsync` is reduced, the overall performance is decreased. It also indicates that more frequent copy and garbage collection increases `fsync` frequency, which results in larger drop in performance. This result demonstrates Ebtrfs has achieved higher performance while keeping the same level of consistency as default btrfs in a wide range of number of write operations per `fsync`.

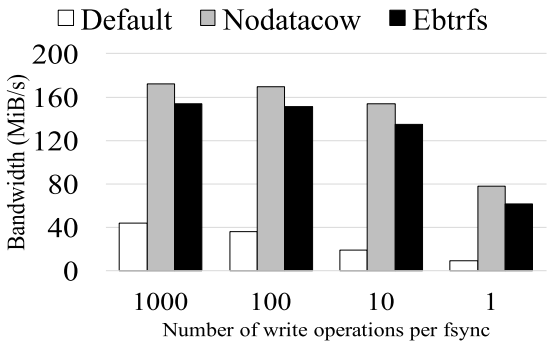


FIGURE 7. FIO results in ebtrfs with different fsync parameters.

TABLE 2. InnoDB experimental parameters.

Parameters	Values
Page size (KiB)	4
DB buffer size (GiB)	1
Flush method	fsync
Doublewrite	ON or OFF
The number of records	100,000,000
Number of clients	16-128

2) OLTP PERFORMANCE

We ran the sysbench OLTP benchmark with MySQL 5.6.21 and InnoDB for e2xt4 and ebtrfs to show a more real application performance. Table 2 shows the experimental parameters and other parameters are configured as the default. We configure the page size as 4 KiB instead of the default page size (16 KiB) since a smaller page size leads to better performance [17]. InnoDB supports a technique for guaranteeing the atomicity by performing redundant writes called Double-Write Buffer (DWB). InnoDB first writes and flushes data to a double write buffer area and then writes and flushes each data to its original location. E2xt4 and ebtrfs can disable this technique because they perform transaction processing within TxSSD.

a: ENHANCED EXT4

Figure 8 depicts OLTP results on ext4 with the ordered mode and e2xt4. Three configurations are chosen, such as ext4

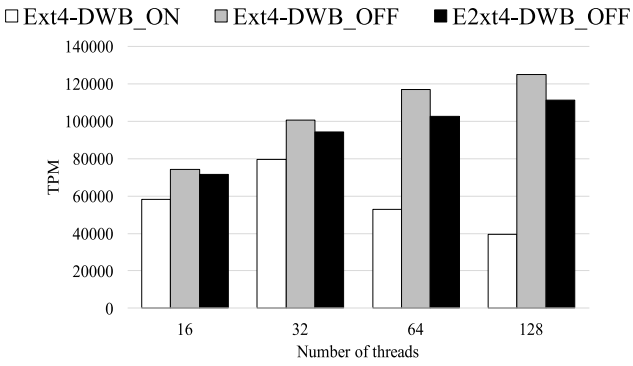


FIGURE 8. OLTP results between ext4 (ordered mode) and e2xt4 file systems.

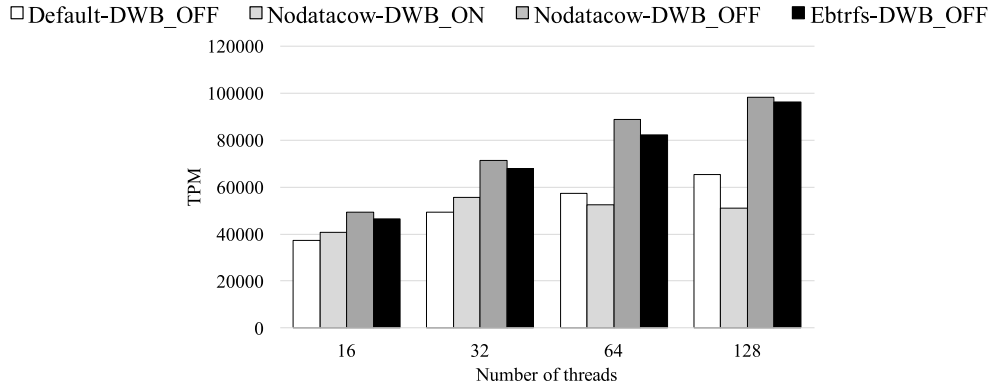


FIGURE 9. OLTP results between btrfs and ebtrfs file systems.

(ordered mode) with or without DWB, and e2xt4 without DWB. As shown in the figure, in the case of the ordered mode with DWB, the TPM increases by 36% from 16 to 32 threads but does not scale after 32 threads. In the case of 64 and 128 threads, the TPM decreases to 52729 (65.9%) and 39384 (49.6%) respectively. The reason of performance reduction is the increased I/O traffic and contention due to DWB. That is, DWB does not have any parallelism and the DWB operation writes to a DWB area sequentially, which harms the scalability. Meanwhile, the performance of ext4 and e2xt4 without DWB scales well since there is no I/O contention on DWB. Ext4 without DWB improves the performance by 2.21X and 3.17X compared to ext4 with DWB in the case of 64 and 128 threads, respectively. The large performance gap shows that enabling DWB incurs a significant overhead in trade with consistency. E2xt4 without DWB improves the performance by 1.23X, 1.19X, 1.95X, and 2.82X in the case of 16, 32, 64, and 128 threads, respectively while providing the same level of consistency as ext4 with DWB. Compared to ext4 without DWB, e2xt4 without DWB shows TPM of 96.3%, 93.7%, 87.9%, and 89.0% with 16, 32, 64, and 128 threads, respectively. This result indicates the transaction processing overhead presented in the TxSSD.

b: ENHANCED BTRFS

Figure 9 shows the OLTP results on btrfs and ebtrfs with four configurations such as default (datacow) without DWB, nodatacow with DWB, nodatacow without DWB, and ebtrfs without DWB. The default mode without DWB and the nodatacow mode with DWB maintain the same level of consistency but with different performance implications. As shown in the figure, the nodatacow with DWB mode shows higher performance than the default without DWB in 16 and 32 threads, respectively. Meanwhile, in the case of 64 and 128 threads, nodatacow with DWB shows 91.3% and 77.9% higher TPM compared to the default btrfs without DWB. This result shows that DWB has a more negative impact on performance than datacow when the number of threads increases.

Nodatacow without DWB outperforms the default mode without DWB and the nodatacow mode with DWB due to no redundant write operations for data. Ebtrfs without DWB outperforms the default without DWB by 1.24X, 1.38X, 1.43X, and 1.49X and it also outperforms nodatacow with DWB by 1.14X, 1.22X, 1.57X, and 1.89X in the case of 16, 32, 64, and 128 threads, respectively. Owing to the transactional processing overhead, ebtrfs without DWB generates an overhead of 6.1%, 4.5%, 7.5%, and 2.2% compared with nodatacow without DWB in the case of 16, 32, 64, and 128 threads, respectively. However, this overhead is much smaller than the overheads generated by datacow or DWB. Consequently, our results demonstrate that ebtrfs keeps the same level of consistency like datacow or DWB with small overhead.

TABLE 3. Recovery time.

File systems	Times (msec)
ext4(ordered)-DWB_ON	249.3
e2xt4(TxSSD)-DWB_OFF	0.8
btrfs(datacow)-DWB_OFF	1379.5
btrfs(nodatacow)-DWB_ON	20.6
ebtrfs(TxSSD)-DWB_OFF	0.7

3) RECOVERY PERFORMANCE

To measure recovery time, we cut the power of the machine while it was executing the OLTP benchmark. Table 3 shows the recovery time after rebooting the machine. The recovery time of e2xt4 is 0.8 ms. Meanwhile the recovery time of the ordered mode with DWB is 249.3 ms. The recovery procedure of the ordered mode performs the scan and replay operations for the metadata. The datacow mode of btrfs increases the recovery time since the datacow mode reconstructs its tree nodes for both data and metadata. Meanwhile, the nodatacow mode of btrfs with DWB reduces the recovery time since the nodatacow mode only recovers the metadata. Similar to e2xt4, the recovery time of ebtrfs is only 0.7 ms. Meanwhile, the recovery time of the datacow mode without DWB and the nodatacow mode with DWB in btrfs is 1379.5 ms and 20.6 ms, respectively. In summary, recovery time of e2xt4 and ebtrfs is the shortest while providing the same level of crash

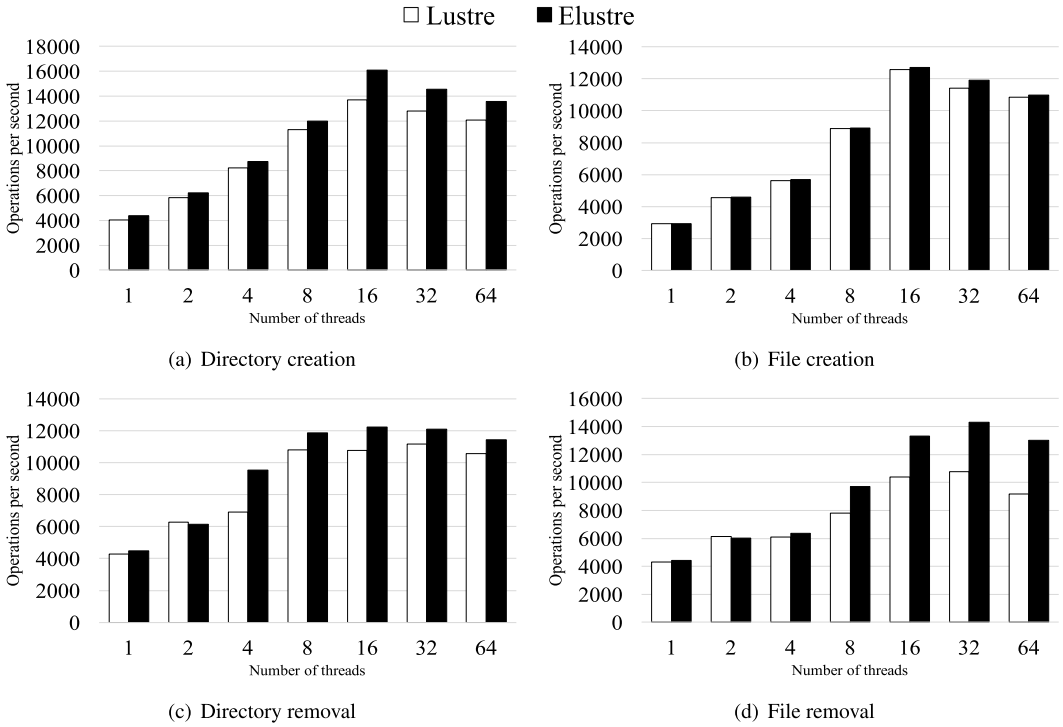


FIGURE 10. Mdtest benchmark results for directory creation, file creation, diretory removal, and file removal.

consistency. The reason is that e2xt4 and ebtrfs only perform the recovery operation to TxSSD in which all mapping information in the uncommitted tx tables is just discarded.

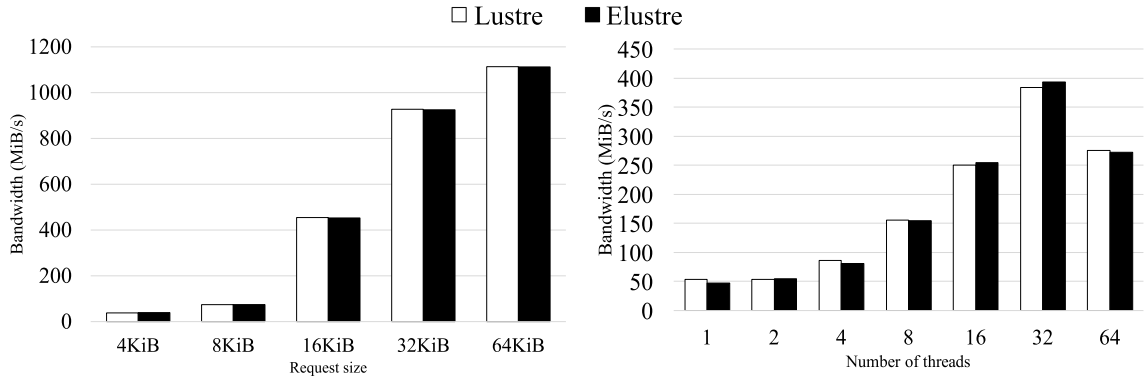
C. DISTRIBUTED PARALLEL FILE SYSTEM PERFORMANCE

We use mdtest and IOR which are widely used to measure the performance of distributed parallel file systems. Mdtest is an MPI-coordinated metadata-intensive benchmark. Each task creates, stats, and removes the specified number of directories and/or files and measures the performance in operations per second. IOR is an MPI-coordinated data-intensive benchmark with various interfaces and access patterns. We configured both IOR and mdtest in various numbers of options and threads. We note that the current existing Lustre file system does not support the data journaling mode due to the performance issue. Thus, we compare our elustre file system with the existing Lustre file system in the ordered mode, and the consistency level of our enhanced Lustre file system is higher than that of the existing Lustre file system.

1) MDTEST PERFORMANCE

We evaluate the metadata I/O performance in existing and enhanced Lustre file systems by using the mdtest benchmark. In our evaluation, we set five branch, five depth, and ten items, which means the mdtest first creates a directory tree with five branch and five depth, and each tree node creates ten items (files or directories). We set the number of bytes to write to each file after it is created as 4 KiB. Figure 10 shows the operations per second for the directory/file creations and the directory/file removal when the number of threads is increased. Figure 10(a) shows the performance of the directory creation operations. As shown in the figure, elustre improves the performance by 8%, 7%, 7%, 6%, 18%, 13%, and 12% compared to Lustre at each number of threads, respectively. For the improvement, the reason is that mdtest generates metadata-intensive operations, and elustre processes the transaction for the metadata without redundant writes while Lustre processes the transaction with redundant writes. Thus, elustre improves the metadata-intensive performance compared to Lustre in the ordered mode. Figure 10(b) shows the performance for file creation operations. The results of the two file systems are similar. When the number of threads is 32, we improve the performance up to 4.4%.

Figure 10(c) shows the performance of directory removal operations. As shown in this figure, elustre improves the performance by 4.3%, 0.3%, 38%, 10%, 13.7%, 8.4%, and 8.3% at each number of threads, respectively, compared to Lustre. Figure 10(d) shows the performance of the file removal operations. Elustre improves the performance by 3.3%, 0.2%, 4.1%, 24.7%, 27.9%, 32.9%, and 42% at each number of threads, respectively, compared to Lustre. When the number of threads is 64, elustre achieves maximum improvement. Consequently, we show a similar or better performance compared to the existing file system while providing a higher consistency level. It is because that the existing file system provides the transactions for the metadata in the MDT and the OST but does not provide the transactions for data for the OST in which writes the data of the file.



(a) Different block sizes (b) Different number of threads

FIGURE 11. IOR benchmark results according to different block sizes and number of threads.

2) IOR PERFORMANCE

We evaluate existing and enhanced Lustre file systems by using the IOR benchmark. Figure 11(a) shows the random write performance with different request sizes when the number of threads is eight and the number of files is one per thread, which means each thread creates a single file with six stripes, and these stripes will be distributed into six OSTs. The overall performance increases as the number of threads increases. but the performance is decreased at 64 threads. As shown in the figure, the performance of the two file systems is similar. It is because that IOR generates data-intensive operations, and most I/O operations are performed in the OSTs with small metadata I/Os in both MDT and OSTs unlike the case of mdtest. Unlike the result of local file systems, the overhead of TxSSD is hidden since many layers and components of Lustre and network overhead generate a longer latency compared to a local file system. In terms of the consistency level, elustre provides a higher consistency level than that of Lustre since Lustre in the ordered mode only supports the transaction for metadata, but our elustre supports the transaction for metadata and data.

Figure 11(b) shows the result of random write in different threads. In this evaluation, we set the request size as 16 KiB. As shown in the figure, the performance of the two file systems is similar as the performance results with different request sizes. When the number of threads is 32, the two file systems show the highest bandwidth. The bandwidth of elustre and Lustre is 393.92 MiB/s and 384.11 MiB/s, respectively. This demonstrates that the overhead of TxSSD is completely hidden, and elustre shows a slightly better performance even with fewer metadata operations. Figure 12 shows the performance of `fsync` per random write operation in the different number of threads. The frequent `fsync` call decreases the overall performance. Meanwhile, the performance increases as the number of threads increases. Similar to other results of the IOR benchmark, the performance of the two file systems is similar. This result shows that unlike the case of local file systems, the `fsync` call affects the performance less in the distributed parallel file system.

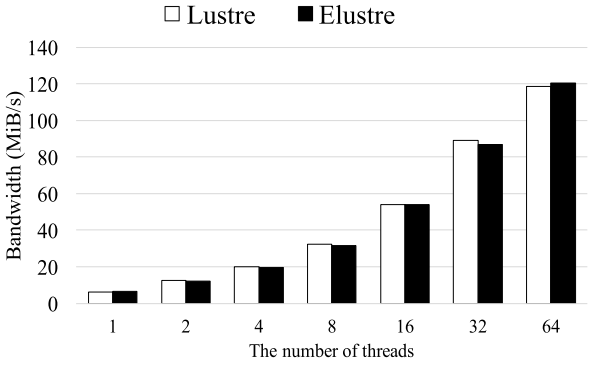


FIGURE 12. IOR benchmark results for fsync operation per write operation on the different number of threads.

D. EXPERIMENTAL ANALYSIS

We disclose the overhead of the transactional support from our file systems without the overhead inside TxSSD. To do this, we disable the transactional functionality of TxSSD. Then, we compare the our modified version with the unmodified version in order to disclose the overhead of our implementation. Table 4 depicts that there is almost no overhead of the transaction support in our file systems. Thus, the overhead is attributed to the overhead of the transactional support generated by the TxSSD.

TABLE 4. Performance of enhanced file systems and modified block I/O subsystem when functionality of TxSSD is disabled.

File systems	Bandwidth (MiB/s)
e2xt4(enable TxSSD)	153
e2xt4(disable TxSSD)	152.5
ebtrfs(enable TxSSD)	153.5
ebtrfs(disable TxSSD)	154.8
elustre(enable TxSSD)	1112.3
elustre (disable TxSSD)	1112.5

To support this claim, we measure the latency of normal and transactional operations with a request size of 4 KiB using one thread. To reduce the overhead of measurement,

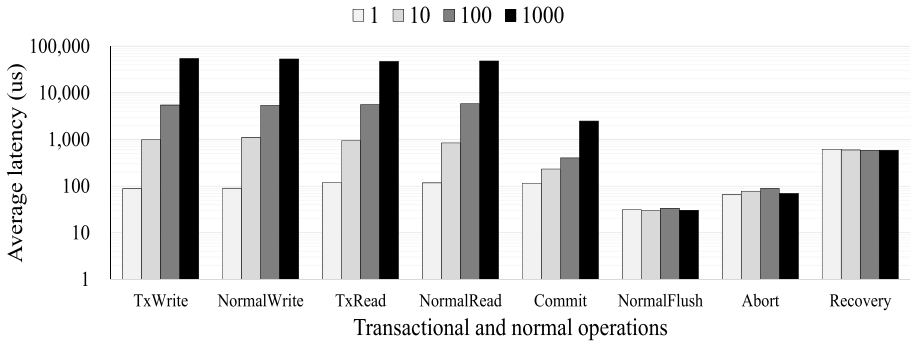


FIGURE 13. Average latency of normal and transactional operations under the number of different entries.

we use a raw device and the direct I/O mode. In the raw device, we use commit-on-flush to commit a transaction with the flush command, which remaps the entries in the tx table to the entries in the FTL mapping table because there is no commit block.

Figure 13 depicts the average latency of transactional operations (TxWrite/Read, Commit, Abort, and Recovery) and normal operations (NoramlWrite/Read/Flush) under different numbers of entries. For instance, 10 means that 10 entries are updated and remapped to the FTL mapping table in TxWrite and the commit operation, respectively. There is almost no performance gap between the TxWrite/Read and NormalWrite/Read. This result demonstrates that update and search operations hardly affect performance in TxSSD.

Meanwhile, the commit operations produce a considerable overhead compared with normal flush. In the SSD firmware, the time taken by the remap function is 2474us, which is similar to the measured result in the host side. In the case of commodity or Open SSDs, the time taken by the flushing command is a few milliseconds. Meanwhile, in the case of our SSD, the flushing command produces about 30 us because it returns instantly due to supercapacitors. Thus, the overhead of remap operations[5] becomes more noticeable to the host. It affects the performance of applications as the number of entries increases.

The latency of recovery operation is longer than that of the abort operation because the recovery operation discards mapping information in all tx tables. Even though the number of entries increases, the latencies for the abort and recovery operations are not increased. The reason is that TxSSD deallocates the tx tables with the uncommitted entries.

V. RELATED WORK

A. FILE SYSTEMS FOR FLASH-BASED SSDs

There are many studies on file systems for flash SSDs. F2FS [20] is a file system designed for flash SSDs. F2FS devises a flash-friendly on-disk layout to avoid unnecessary data copying and multi-head logging for optimizing the write performance. ParaFS [47] is a log-structured file system for flash SSD to exploit internal parallelism inside SSD while ensuring efficient garbage collection. ParaFS coordinates the garbage collection at both the file system and FTL levels. It also schedules read, write, and erase requests over multiple channels to achieve consistent performance.

SpanFS [16] is a scalable file system for flash-based SSDs. SpanFS consists of a collection of micro file system services called domain to improve the scalability of file systems on many cores. It distributes files and directories among the domains and provides a global file system view on top of the domains. This article is in line with the previous works [16], [20], [47] in terms of improving the performance of file systems based on flash-based SSDs. In contrast, we focus on improving and evaluating the performance of file systems on flash-based SSDs that support the transaction functionality.

B. DISTRIBUTED PARALLEL FILE SYSTEMS

There have been several studies on distributed parallel file systems. Devulapalli and Wycoff analyze strategies for file creation in file systems that distribute metadata across multiple servers. They present designs, which reduce the message complexity of the create operation and increase the performance. IndexFS [35] provides scalable high-performance operations on the metadata and small files for existing file systems, such as PVFS, Lustre, and HDFS. IndexFS uses a table-based architecture and an optimized log-structured layout that stores the metadata and small files efficiently.

PLFS [6] is a parallel log structured file system. PLFS remaps an application's preferred data layout into one which is optimized for the underlying file system. The layer of indirection and reorganization reduces the checkpoint time. Piernas et al. [31] propose a user-space implementation of active storage for Lustre and compare it with the traditional kernel-based implementations. They show that the user-space approach prove to be faster, more flexible, portable, and deployable than the kernel-space approaches.

Oral *et al.* [29] found that journaling in Lustre for the object store considerably affects the overall performance. To increase the overall performance of the file system, they provide a hardware solution using external journaling devices and propose software-based optimization to remove

[5]The remap function performs copy operations (memory) for the entries from the tx table to the FTL mapping table.

the synchronous commit. Our work is in line with these studies [6], [11], [29], [31], [35] in terms of investigating the distributed parallel file systems and their performance. In contrast, we focus on improving the performance using TxSSDs and investigate their implications.

C. TRANSACTIONAL SSDs

There are many studies for supporting transactional functionality in SSDs. The transactional SSD concept was first introduced by TxFlash [33], which provides cyclic commit protocols. TxFlash links all pages in each transaction in one cyclic list by keeping pointers in the page metadata. The cyclic commit uses per-page metadata to remove the need for a separate commit record. It requires judging whether a transaction is committed or not. Meanwhile, this scheme can be inappropriate for current enterprise SSDs. The reason is that the metadata area can be reserved for multiple purposes to contain information about error correcting code (ECC) data, bad blocks, etc [42].

LightTx [22], [23] supports transaction flexibility using a lightweight embedded transaction design. LightTx uses a commit protocol that determines the transaction state solely inside each transaction in order to support parallel transaction execution. In addition, LightTx periodically retires the dead transactions to reduce transaction state tracking cost. DiffTx [24] is an embedded transaction protocol which differentially logs partial page updates in a write-ahead logging way and writes full page updates in a shadow paging way, aiming at low write amplification. TxCache [25] is a new embedded transaction mechanism for SSDs with non-volatile disk cache. TxCache design leverages non-volatile disk cache to efficiently support transactions inside SSDs. It persists new-version data in non-volatile disk cache in a shadow way while protecting old-version data from being overwritten. LightTx, DiffTx, and TxCache are design and implemented on a trace-driven SSD simulator based on DiskSim. In contrast with these studies, we aim to show the experimental results by using SSDs which satisfies enterprise and standard interface.

Shi *et al.* [39] is a transactional SSD design which provides different types of transactional primitives to support static and dynamic transactions separately. Mobius flash translation layer (mFTL) combines normal FTL with transaction processing by storing mapping and transaction information together in a physical flash page by using out-of-band (OOB). In this case, this design may not adapt to other flash-based SSDs. It is because that OOB area can be used for error correction code (ECC) and metadata by each vendor. Mobius is designed and implemented on openSSD platform which is not enterprise-class SSD. The one of main differences point between OpenSSD and enterprise SSD is whether the supercapacitor is supported or not. In the case of enterprise SSD, the supercapacitor is supported so that the flush command overhead is very low. Thus, the results of evaluation can be totally difference between OpenSSD and enterprise SSD. This paper discloses the evaluation results and analyze the results.

FusionIO [30] presents an atomic-write in an enterprise flash-based SSD. It provides *atomic write* that puts a batch of multiple I/O operations into a single logical group, which is persisted successfully or rolled back upon a failure. FusionIO modifies MySQL to use the atomic write call. Meanwhile, there is no performance evaluation on file systems. We also focus on the performance for transactional SSD based on SATA and the firmware FTL rather than a host-based FTL that consumes host resources.

X-FTL [18] improves the performance of SQLite by exploiting the transactional atomicity provided by SSDs. SQLite is a DBMS used by Android phones, which relies on costly page-oriented journaling to support transactional atomicity. This results in slow responses to mobile applications. CFS [27] is a file system built on X-FTL [18]. CFS guarantees crash consistency in application level by enabling applications to declare arbitrary code regions which is required for providing crash consistency. CFS improves performance of SQLite and solves the problem of false sharing of metadata. Unfortunately, their [18], [27] performance evaluation is based on openSSD [3], which is not an enterprise-class SSD.

SHARE interface [28] provides an abstraction which allows applications to change the address mapping inside flash storage. It allows the applications to achieve write atomicity without write amplification. The goal of the previous studies [18], [22]–[24], [27], [28], [30], [33], [39] is similar to our goal in terms of supporting transactional functionality inside SSD. Meanwhile, to the best of our knowledge, this work is the first evaluation study on transactional SSDs that satisfies both enterprise and SATA interface to provide performance results for a more realistic and standard fashion. Furthermore, we evaluate the effectiveness of transactional SSDs in a distributed parallel file system, as well as local file systems.

VI. SUMMARY AND IMPLICATION

We summarize implications of our evaluation study. Especially, we make the following main findings and insights throughout our measurements and observations.

- Redundant writes for providing crash consistency affect the performance and endurance, especially in btrfs, while the overhead from flush operations is small due to the supercapacitor.
- The overhead of memory copy operations inside TxSSD during the commit operation is noticeable because the overhead of the flush command is low. Except for the commit operations, the latency of other transactional operations on TxSSD is the same as those on the original SSD.
- E2xt4 and ebtrfs on TxSSD improve I/O and recovery performance compared with the original ext4 and btrfs while providing the same consistency level. However,

the I/O performance in the data-intensive workloads is slightly lower than that of the weak-consistent modes (ordered and nodatacow) due to the overhead of transactional support inside TxSSD.

- Furthermore, elustre with multiple TxSSDs improves the I/O performance in the metadata-intensive workload and shows a similar performance in the data-intensive workload but provides a higher consistency level compared with Lustre.

VII. CONCLUSION

In this article, we investigated the implications of transactional SSDs. We evaluated the effect of transactional SSDs with diverse file systems and configurations. Then we found insights and observations through our evaluation. Our results show that TxSSD-aware file systems increase the performance compared with crash-consistent modes while maintaining crash consistency. Additionally, they improve the endurance of SSDs because the total number of writes decreases. Finally, we show that providing transactional functionality in SSDs may incur overhead. It must be considered carefully when we design transactional functionality or in-storage computing.

REFERENCES

[1] *IOR HPC Benchmark*. Accessed: Sep. 1, 2019. [Online]. Available: https://sourceforge.net/projects/ior-sio/

[2] *Mdtest: HPC Benchmark for Metadata Performance*. Accessed: Sep. 1, 2019. [Online]. Available: http://sourceforge.net/projects/mdtest/

[3] *The Openssd Project*. Accessed: Sep. 1, 2019. [Online]. Available: http://www.openssd-project.org/wiki/The_OpenSSD_Project

[4] I. Polato, R. Ré, and F. Kon, "A comprehensive view of Hadoop research—A systematic literature review," *J. Netw. Comput. Appl.*, vol. 46, pp. 1–25, Nov. 2014.

[5] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 52–54, Jul. 2010.

[6] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, p. 21.

[7] J. Bonwick and B. Moore, "ZFS: The last word in file systems," Sun Microsyst., Tech. Rep., 2007.

[8] V. Chang and G. Wills, "A model to compare cloud and non-cloud storage of Big Data," *Future Gener. Comput. Syst.*, vol. 57, pp. 56–76, Apr. 2016.

[9] D. A. Solomon, *Inside Windows NT* (Microsoft Programming Series). Seattle, WA, USA: Microsoft Press, 1998.

[10] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux J.*, vol. 2013, no. 235, p. 1, 2013.

[11] A. Devulapalli and P. Wyckoff, "File creation strategies in a distributed metadata file system," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10.

[12] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 1992.

[13] H. Reiser. (2004). *Reiserfs*. [Online]. Available: http://www.namesys.com

[14] J. Axboe. (Apr. 1998). *Fiobenchmark*. [Online]. Available: http://freecode.com/projects/fio

[15] (2002). *JFS for Linux*. [Online]. Available: http://oss.software/ibm.com/jfs

[16] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai, "Spanfs: A scalable file system on fast storage devices," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, 2015, pp. 249–261.

[17] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable write cache in flash memory SSD for relational and NOSQL databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2014, pp. 529–540.

[18] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2013, pp. 97–108.

[19] (Sep. 1, 2019). *SysBench: A System Performance Benchmark*. [Online]. Available: http://sysbench.sourceforge.net

[20] 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. (FAST)*, 2015, pp. 273–286.

[21] Y.-S. Lee, L. C. Quero, S.-H. Kim, J.-S. Kim, and S. Maeng, "Activesort: Efficient external sorting using active SSDs in the mapreduce framework," *Future Gener. Comput. Syst.*, vol. 65, pp. 76–89, Dec. 2016.

[22] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTX: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 115–122.

[23] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "High-Performance and Lightweight Transaction Support in Flash-Based SSDs," *IEEE Trans. Comput.*, vol. 64, no. 10, pp. 2819–2832, Oct. 2015.

[24] Y. Lu, J. Shu, J. Guo, and P. Zhu, "Supporting system consistency with differential transactions in flash-based SSDs," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 627–639, Feb. 2016.

[25] Y. Lu, J. Shu, and P. Zhu, "TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Aug. 2014, pp. 1–6.

[26] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, Ottawa, ON, Canada, 2007. [Online]. Available: http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf

[27] C. Min, W.-H. Kang, T. Kim, and S.-W. Lee, "Lightweight application-level crash consistency on transactional flash storage," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 221–234.

[28] G. Oh, "SHARE interface in flash storage for relational and NoSQL databases," in *Proc. SIGMOD*, New York, NY, USA, 2016, pp. 343–354.

[29] S. Oral, F. Wang, D. Dillow, G. M. Shipman, R. Miller, and O. Drokin, "Efficient object storage journaling in a distributed parallel file system.," in *Proc. FAST*, 2010, pp. 143–154.

[30] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2011, pp. 301–311.

[31] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2007, Art. no. 28.

[32] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 2005, pp. 105–120.

[33] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Conf. Operating Syst. Design Implement. (OSDI)*, Berkeley, CA, USA, 2008, pp. 147–160.

[34] V. R. Chandakanna, "REHDFS," *J. Netw. Comput. Appl.*, vol. 103, pp. 85–100, Feb. 2018.

[35] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. SC14: Int. Conf. for High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 237–248.

[36] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The Linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, 2013, Art. no. 9.

[37] 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, 1992.

[38] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proc. Linux Symp.*, 2003, pp. 380–386.

[39] W. Shi, D. Wang, Z. Wang, and D. Ju, "Mobius: A high performance transactional SSD with rich primitives," in *Proc. 30th Symp. Mass Storage Syst. Technol. (MSST)*, Jun. 2014, pp. 1–11.

[40] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST)*, Washington, DC, USA, May 2010, pp. 1–10.

[41] 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.

[42] STMicroelectronics. (2014). *Stmicroelectronics NAND ECC Schemes*. [Online]. Available: http://www.stlinux.com/howto/NAND/ST-ECC

[43] H. Sun, G. Chen, J. Huang, X. Qin, and W. Shi, "CalmWPC: A buffer management to calm down write performance cliff for NAND flash-based storage systems," *Future Gener. Comput. Syst.*, vol. 90, pp. 461–475, 2018.
[44] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. USENIX Annu. Tech. Conf.*, vol. 15, 1996.
[45] S. C. Tweedie, "Journaling the Linux ext2fs filesystem," in *Proc. 4th Annu. Linux Expo*, 1998.
[46] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Design Implement.*, 2006, pp. 307–320.
[47] J. Zhang, J. Shu, and Y. Lu, "Parafs: A log-structured file system to exploit the internal parallelism of flash devices," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*. Denver, CO, USA: USENIX Association, 2016, pp. 87–100.

YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science, Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana–Champaign. He is currently an Assistant Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interests include operating, distributed, and database systems.

HEON YOUNG YEOM received the B.S. degree in computer science from Seoul National University, in 1984, and the M.S. and Ph.D. degrees in computer science from Texas A&M University, in 1986 and 1992, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Seoul National University. His research interests include database systems and distributed systems.

HYUCK HAN received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, Seoul, South Korea, in 2003, 2006, and 2011, respectively. He is currently an Assistant Professor with the Department of Computer Science, Dongduk Women's University. His research interests include distributed computing systems and algorithms.