DJFS: Providing Highly Reliable and High-Performance File System with Small-Sized NVRAM

Junghoon Kim, Minho Lee, Yongju Song, and Young Ik Eom

File systems and applications try to implement their own update protocols to guarantee data consistency, which is one of the most crucial aspects of computing systems. However, we found that the storage devices are substantially under-utilized when preserving data consistency because they generate massive storage write traffic with many disk cache flush operations and force-unit-access (FUA) commands. In this paper, we present DJFS (Delta-Journaling File System) that provides both a high level of performance and data consistency for different applications. We made three technical contributions to achieve our goal. First, to remove all storage accesses with disk cache flush operations and FUA commands, DJFS uses small-sized NVRAM for a file system journal. Second, to reduce the access latency and space requirements of NVRAM, DJFS attempts to journal compress the differences in the modified blocks. Finally, to relieve explicit checkpointing overhead, DJFS aggressively reflects the checkpoint transactions to file system area in the unit of the specified region. Our evaluation on TPC-C SQLite benchmark shows that, using our novel optimization schemes, DJFS outperforms Ext4 by up to 64.2 times with only 128 MB of NVRAM.

Manuscript received Aug. 12, 2016; revised Mar. 14, 2017; accepted Sept. 27, 2017.

Junghoon Kim (myhuni20@skku.edu) is with the Department of IT Convergence, Sungkyunkwan University, Suwon, Rep. of Korea.

Minho Lee (minhozx@skku.edu) and Yongju Song (eluphany@skku.edu) are with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Rep. of Korea.

Young Ik Eom (corresponding author, yieom@skku.edu) is with the College of Software, Sungkyunkwan University, Suwon, Rep. of Korea.

This is an Open Access article distributed under the term of Korea Open Government License (KOGL) Type 4: Source Indication + Commercial Use Prohibition + Change Prohibition (http://www.kogl.or.kr/news/dataView.do?dataIdx=97).

Keywords: High-performance, Journaling file system, NVRAM, Reliability.

I. Introduction

Ensuring data consistency and durability is one of the most crucial aspects of computing systems. To guarantee system-wide consistency of file system data, a variety of techniques have been proposed and deployed [1]–[4]. For example, under Ext4, system-wide consistency is guaranteed by first writing updated blocks to the journal area during the commit phase and then writing them to their original locations (that is, home locations) during the checkpoint phase. If a sudden power loss or system failure occurs, we can restore the file system to a consistent state by using the journal data. Moreover, to guarantee application-level consistency of critical data, applications implement their own update protocols (for example, Rollback and WAL modes of SQLite) [5].

However, we found that storage devices are substantially under-utilized when preserving system-wide and application-level consistency. First, Ext4 writes the updated blocks twice, once in the journal area and once in the file system area, for system-wide consistency. This can increase the storage write traffic, and thus degrades the system performance and shortens the lifespan of the storage devices. In addition to duplicate writes, Ext4 invokes a disk cache flush operation with a force-unit-access (FUA) command at the end of each of commit and checkpoint phase to guarantee durability and the correct ordering of updates. Unfortunately, these frequent disk cache flush operations with FUA commands degrade the system performance more seriously because the storage firmware flushes all cached data in the DRAM write-cache to non-volatile media upon the disk cache flush operation.

 © 2017 ETRI

Second, the update protocols of applications frequently trigger fsync() system calls for application-level consistency [6]. The frequent fsync() calls generate a massive amount of random write traffic with many disk cache flush operations that must be handled synchronously [7]. In particular, each fsync() call blocks the application until completion of the disk cache flushing, thereby causing a non-trivial delay in the application performance. To optimize the overall system performance, some file systems, device drivers, and virtual machines deliberately disable the disk cache flush commands [8]. However, this cannot ensure the durability of the updates, and can therefore lead to inconsistency in the file system. In addition, to relieve the file system journaling overhead, most computing systems adopt ordered journaling mode, which writes only the file system metadata to the journal area. However, this can lead to inconsistencies in the file data upon system recovery, even if the file system carefully orders the writing of file data and file system metadata [9]. Furthermore, even if the computing system adopts ordered journaling mode for the system performance, update protocols of the applications impose a significant performance overhead owing to the frequent fsync() calls. In this paper, we propose DJFS (Delta-Journaling File System) that provides both high reliability and a high level of performance for different applications. Our technical contributions to DJFS are as follows:

Journaling without Flushes: Based on several analyses, we found that Ext4 periodically generates journal write traffic with disk cache flush operations and FUA commands. Furthermore, when applications are designed to exploit fsync() for application-level consistency, the fsync() calls incur excessive journaling overhead. Thus, even if storage devices embed a great deal of DRAM write-cache in order to optimize the storage performance, the runtime performance is eventually bounded by the speed of the disk cache flushing. DJFS removes all storage accesses with disk cache flush operations and FUA commands upon periodic commit operations or frequent fsync() calls, while providing system-wide consistency.

Delta Journaling: We found experimentally that the use of NVRAM journal with full journal mode does not achieve the maximum performance owing to frequent check-pointing overhead. To relieve this problem caused by a lack of free journal area, we introduced a novel mechanism for DJFS, called Delta Journaling, which attempts to journal compress the differences in the modified blocks, rather than the full blocks themselves, while efficiently exploiting the byte-addressable characteristics of NVRAM. We also found that, when the journaling is triggered owing to an fsync() call, journal metadata such as the journal descriptor (JD) and journal commit (JC) consume only a few bytes, although they are inefficiently journaled as a full block. Thus, to reduce the volume of journal metadata writes, DJFS system only writes to the used space of JD and JC.

Aggressive Group Checkpointing: Compared to the original journaling, DJFS does not issue write requests to the storage device upon periodic commit operations or frequent fsync() calls by using small-sized NVRAM efficiently as a journal area. Thus, DJFS has an opportunity to exploit the idle time of the storage device. To further relieve the overhead of frequent checkpointing, we introduce an Aggressive Group Checkpointing scheme that aggressively reflects the checkpoint transactions to the file system area in the unit of the specified region. Using this optimization scheme, we relieve the overhead of stalled writes, and can thus further improve the application performance.

We implemented DJFS based on Ext4. The evaluation results with various benchmarks clearly show that DJFS significantly improves the performance by up to 64.2 times by dramatically reducing the journal write traffic along with the number of disk cache flush operations and FUA commands through our novel optimization schemes.

The remainder of this paper is organized as follows. Section II describes the background on the concepts and issues regarding the consistency. The consistency overhead and various related aspects are analyzed in Section III. We discuss our design principles in Section IV, and present the details of DJFS design in Section V. We show our evaluation results in Section VI. Finally, we present related works in Section VII, and conclude our study in Section VIII.

II. Background

1. System-Wide Crash Consistency

File systems use a variety of techniques such as journaling [3], [4] and copy-on-write [1], [2] to guarantee system-wide crash consistency. Of these, we describe Ext4, which is the basis of DJFS, in detail.

Ext4 provides three journal modes: writeback, ordered, and full. For performance reasons, the writeback and ordered modes journal only the file system metadata to the journal area, and thus file data consistency is not guaranteed after a system crash. Unlike writeback mode, data ordering is preserved in ordered mode. This means that dirty data blocks are flushed to the file system area before the metadata blocks are written to the journal area. The full journal mode logs all file data and metadata

blocks to the journal area. This mode provides system-wide consistency, but generates a huge performance overhead owing to the bulky duplicate writes. Thus, all high-performance production file systems including Ext4 only provide metadata consistency by default, even though this can lead to inconsistencies in the file data upon a system crash.

Ext4 communicates with the JBD2 journaling layer in order to conduct the commit and checkpoint operations. Specifically, a commit operation writes the updated blocks to the journal area periodically (for example, in 5 s intervals), or at an explicit sync request such as an fsync() call, for potential use during a system recovery. After an unclean shutdown, JBD2 replays the journal data in order to restore the Ext4 to a consistent state. The checkpoint operation updates the file system with the committed data. We briefly describe the step-by-step procedure of Ext4 journaling, assuming that full journal mode is used.

To guarantee the atomic writing of multiple blocks, JBD2 manipulates three types of transaction lists: running, commit, and checkpoint transactions. A running transaction maintains a list of dirty blocks after the previous commit operation. Whenever a commit operation is triggered, the running transaction is converted into a commit transaction, and the dirty blocks of the commit transaction are targeted for journaling. The *jbd2* kernel thread first writes a JD block to the journal area, as shown in Fig. 1. The JD block starts with a 12-byte journal header, which contains a magic number, header type, and transaction number identifying the logged transaction. The journal block tags follow the journal header, and describe the original locations of the journal blocks that follow in the journal. Although JD may consume a minimum of 36 bytes, it uses a full block inefficiently. The *jbd2* kernel thread then writes the dirty blocks to the journal area. Meanwhile, if new block writes are performed during the commit operation, JBD2 adds them to a new running transaction for the next commit phase. After all dirty blocks are committed, the *jbd2* kernel thread writes a JC block with a WRITE_FLUSH_FUA request to the journal area to identify the end of the transaction. This means that a disk cache flush operation precedes the writing of JC, and the JC block is guaranteed to be on non-volatile media upon completion. The JC block contains the journal header, checksum, and commit timestamp. The JC consumes 32 bytes, but also uses a full block inefficiently. JBD2 then changes the state of the commit transaction into a checkpoint transaction. Through this procedure, the atomicity and durability of a commit operation are guaranteed.

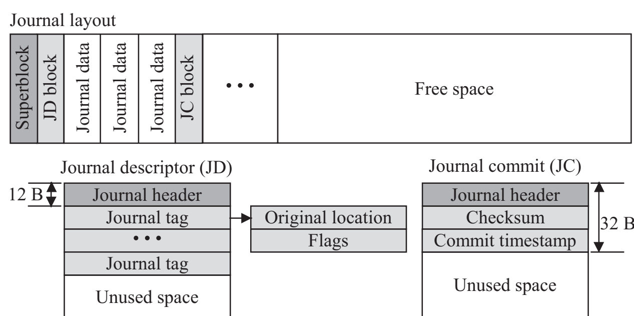

Fig. 1. Layout of the journal and data structures of JD and JC.

After the journal commit is completed, the dirty blocks are exposed to virtual memory; after their dirty timers expire, the blocks are written back asynchronously to the file system through the writeback thread. The *jbd2* kernel thread periodically checks and removes the reflected checkpoint transactions. Meanwhile, if a significant amount of journal space related to the reflected checkpoint transactions can be freed, the *jbd2* kernel thread implicitly reclaims the journal space and updates the journal superblock with a WRITE_FUA request. We call this operation implicit checkpointing. Unfortunately, if the journal area does not have enough free space to handle the upcoming write requests, the explicit checkpointing is triggered. It induces that *jbd2* synchronously reflects the file system metadata or data in checkpoint transactions into the file system with disk cache flush operations and updates the journal superblock with WRITE_FUA requests, until the minimal necessary journal space is reclaimed for the new transaction [10], [11]. Especially, since the explicit checkpointing is frequently triggered in the write-intensive workloads, the runtime performance of the system can be degraded in that application writes are stalled during the reclamation of journal space.

2. Application-Level Crash Consistency

Applications are designed to keep their data crash-consistent when preserving the consistency of application data is crucial. We briefly describe the update protocol of SQLite, which is used to ensure application-level crash consistency.

SQLite provides Rollback and WAL modes to support the atomicity of a transaction execution. In Rollback journal mode (for example, DELETE, TRUNCATE, and PERSIST), the original contents are copied to a rollback journal file before updating them in the database, and thus the changes can always be undone if the transaction aborts. To guarantee the durability and correct ordering of each committed transaction, two fsync() calls are first

https://doi.org/10.4218/etrij.17.0117.0558

invoked for synchronizing the original contents and the journal header to the rollback journal file, and another fsync() call is invoked for synchronizing the updated database file [12]. In WAL journal mode, the original contents are preserved in the database file and the updates are first appended to a separate log file, and any committed changes can therefore be redone by copying them from the log file. Even though WAL mode can decrease the sync overhead compared to Rollback journal mode, WAL journal mode also invokes a fsync() call in each transaction update or log file checkpointing.

Frequent fsync() calls for application-level consistency incur excessive file system journaling, and thus create a huge performance overhead as follows. First, frequent disk cache flushing makes I/O scheduling meaningless owing to fine-grained I/O processing. Second, they unnecessarily increase the storage write traffic because the underlying file system always issues a full block I/O to the storage device, even if a small portion of the block is updated. Furthermore, the overhead of the copying journal metadata such as JD and JC also increases owing to frequent journaling. Finally, the application is blocked during every disk cache flushing, and thus suffers a non-trivial delay quite frequently. For these reasons, there is a basic trade-off between data consistency and high performance when designing and implementing an application.

III. Motivating Experiments

To analyze the overhead for ensuring data consistency, we measured the write IOPS, the number of disk cache control operations including disk cache flushes and FUA commands, and the additional storage writes for journaling, using the FIO benchmark [13]. In our experiments, to understand the system-wide and application-level consistency overhead, the frequency of the fsync() calls was varied from never to once every 64 writes, (see Section VI for a detailed description of the FIO configuration). Ext4 is used as a file system that provides system-wide consistency, and a Samsung 840 SSD is used as a secondary storage device.

First, we found that the journaling technique generates additional journal write traffic for potential use during a system recovery, as shown in Fig. 2(d). Moreover, to guarantee the durability and correct ordering of the updates, it issues disk cache flush operations and FUA commands during the commit and checkpoint operations, as shown in Figs. 2(b) and 2(c). In particular, owing to massive journal writes for file system metadata and file data, Ext4 full journal mode degrades the write IOPS by

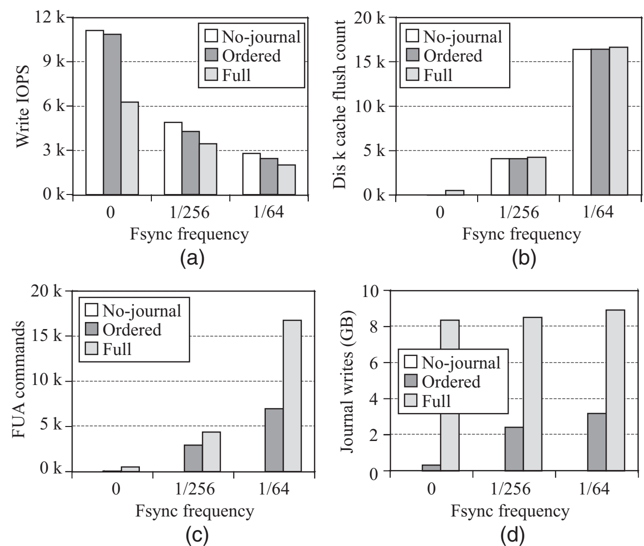

Fig. 2. Analysis of overhead for ensuring data consistency: (a) write IOPS, (b) disk cache flushes, (c) FUA commands, and (d) Journal writes.

up to 43.6% when compared to no journal mode, as shown in Fig. 2(a).

Second, we found that the number of disk cache flush operations and FUA commands increases dramatically in both ordered and full journal modes when the frequency of fsync() increases, as shown in Figs. 2(b) and 2(c). This is because frequent fsync() calls trigger a commit operation within a very short interval. Furthermore, we also found that the excessive commit operations seriously increase the overhead of file system metadata journaling and journal metadata writes such as JD and JC blocks (that is, a Journaling of Journal (JOJ) anomaly), as shown in Fig. 2(d), and thus degrade the write IOPS of the storage device drastically in both Ext4 ordered and full journal modes, as shown in Fig. 2(a). In addition, the write IOPS of no journal mode is seriously affected by frequent fsync() calls because no journal mode issues a disk cache flush operation per fsync() in our evaluation kernel, as shown in Fig. 2(b). From the experiments, we found that update protocols for application-level crash consistency incur a huge performance overhead because fsync() calls are heavily used to maintain their transactions [14].

IV. Design Principles

The design of DJFS begins with a simple question: *How can the file system provide both high performance and strong data consistency for an application*? In this section, we discuss three differentiated design principles to achieve our goal.

File System-Level Approach: To reduce the performance overhead while preserving the data consistency, various changes have been made in the I/O stack, including an update protocol of the applications, the file system, and the storage firmware [6], [15]. However, these solutions cannot be broadly applied to diverse computing systems because they are optimized for specific environments. To obtain a general solution, we designed and implemented DJFS based on Ext4, which is widely adopted in modern computing systems, ranging from a battery-powered smartphone to an enterprise server. Furthermore, we directly exploit file system-level semantics only, and thus do not require any modifications to the applications, device drivers, or storage firmware.

Ensuring Data Consistency: In terms of system-wide consistency, most computing systems adopt ordered journaling mode, even though this can lead to inconsistencies in the file data upon a system crash. This is because the full journal mode seriously increases the storage write traffic, and thus decreases the overall system performance. Moreover, in terms of application-level consistency, update protocols of the applications incur a huge performance overhead owing to the frequent fsync() calls. DJFS aims to provide both high performance and data consistency as the full journal mode in the system level and mitigate the overhead of application-level consistency in the file system level.

Exploiting NVRAM: Traditionally, non-volatile DRAM (NV-DRAM) has been widely used as a write-cache of a storage device in order to enhance the write response time. Now, owing to recent advances in NVRAM technologies, such as phase-change memory (PCM) and spin-transfer torque magnetic RAM (STT-MRAM), they are being considered as a replacement for the main memory [16] or file system storage [17], [18]. However, it is difficult to completely replace the main memory or file system storage with NVRAM owing to its limited density and high cost, although it provides low latency comparable to DRAM. In this regard, we devoted significant effort to find an efficient way to exploit small-sized NVRAM.

V. System Design

1. Journaling without Flushes

A simple solution to relieve the consistency overhead is to use external journaling. Previous research [15] has shown the effectiveness of external journaling in Android mobile devices. Ext4 has an option of journaling updated blocks on a separate block device. Through external journaling, we can split the storage writes into two groups of journal writes and file system writes, and can thus prevent a situation in which heavy I/O loads are concentrated into a single storage device. Furthermore, we can remove randomness in the aggregate traffic by separating the file system and journal I/Os; the locality in the file system area is random, whereas that in the journal area is sequential.

To verify the effectiveness of external journaling, we conducted experiments with the FIO benchmark on two Samsung 840 SSDs; one is for the file system device and the other is for the journal device. In addition, Ext4 is mounted with full journal mode to provide system-wide consistency. As shown in Fig. 3(a), in the case of no fsync() mode, external journaling improves the write IOPS by 46.1% compared to the conventional full journal mode of Ext4; both are full journaling, but one uses an external device for the journal area. However, external journaling does not gain an outstanding achievement in the case of fsync() mode because the I/O performance of external journaling is eventually bounded by the speed of the disk cache flushing. Through our experiments, we found that external journaling is a viable option for system-wide consistency, but cannot relieve the application-level consistency overhead.

DJFS adopts the full journal mode of Ext4 and uses small-sized NVRAM for a file system journal area. This

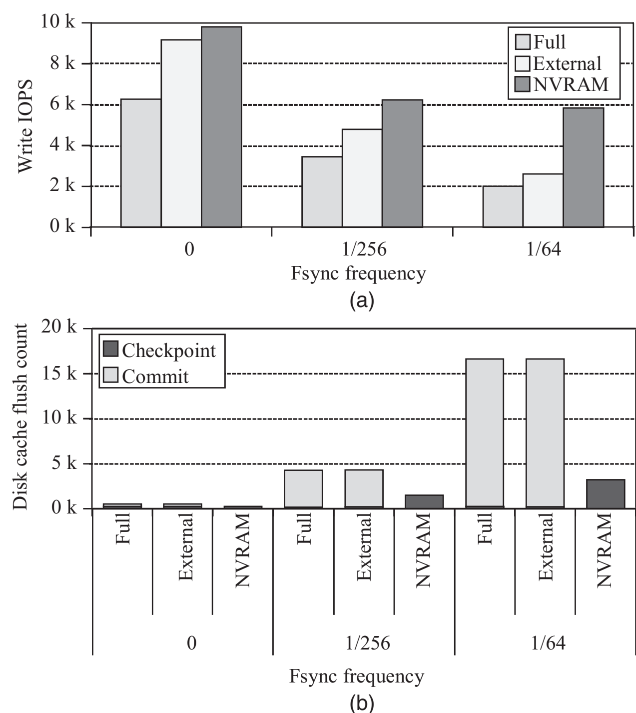

Fig. 3. Experimental results of external and NVRAM journaling: (a) write IOPS and (b) disk cache flushes.

https://doi.org/10.4218/etrij.17.0117.0558

can remove all storage accesses with disk cache flush operations and FUA commands upon periodic commit operations or frequent fsync() calls, while providing system-wide consistency. Of all types of NVRAM, NV-DRAM and STT-MRAM are adequate for our journal device because they promise similar access latency and endurance as DRAM [19]. We also verified the effectiveness of NVRAM journaling with the FIO benchmark. As shown in Fig. 3(a), our first optimization scheme achieves a performance improvement of up to 2.2 times when compared to external journaling. In particular, this scheme gains outstanding achievements in the case of fsync() mode because it can mitigate the overhead of frequent fsync() calls. However, we found that there is some room for a performance improvement in NVRAM journaling. Owing to fast NVRAM journaling, explicit checkpointing is frequently triggered to make free space in the journal area, as shown in Fig. 3(b), and thus NVRAM journaling does not achieve its maximum performance. To further optimize the I/O performance, we introduce two novel techniques in the following subsections.

2. Delta Journaling

The first approach to relieve the overhead of explicit checkpointing is to reduce the size of the journal data. This is a viable solution because file system updates are mostly very small when compared to the size of an entire block [20]. For example, file system metadata are frequently updated, and only a few bytes of a block are actually modified in such cases. Thus, to reduce the space requirements of NVRAM journal, DJFS attempts to journal the compressed differences of the modified blocks (that is, delta), rather than the entire blocks themselves, while exploiting the byte-addressable and non-volatile characteristics of NVRAM. This optimization scheme, called Delta Journaling, also reduces the access latency of NVRAM because it writes significantly smaller amounts of data in the NVRAM [21]. However, DJFS cannot log all journal data in a delta form into the NVRAM journal because each dirty block must first be logged as a form of the entire block in order to guarantee a recovery from a system failure. Furthermore, this may increase the memory overhead because all dirty blocks are involved in difference capturing and delta compression to make a delta form. Thus, DJFS journals each dirty block in the form of an entire block to the NVRAM journal first, and if multiple commit operations are issued to the same dirty block, the dirty block is then logged in a delta form. To do so, each dirty block in the buffer cache is represented with two state indicators, a normal state and a delta state. In the following, we describe how DJFS manages the state of each dirty block in detail.

Figure 4 shows the workings of DJFS during the commit phase. Whenever a system call modifies the file system data, the operating system fetches related blocks to the buffer cache in order to minimize the number of storage access operations. DJFS then initializes each block using a normal state to initially log the journal data in the form of the entire block. After a commit operation is triggered, DJFS first writes a JD to the journal area. For Delta Journaling, we add two fields, the offset and length of the journal data, in the journal block tag.

In addition, to reduce the volume of the journal metadata, DJFS writes only the data of the JD block, excluding the unused space in the block. DJFS then attempts to write the dirty blocks, such as file system metadata and file data, to the NVRAM journal. At this point, DJFS determines the journaling mode of each dirty block. If the dirty block is in a normal state, DJFS writes the entire block to the NVRAM journal, and simply converts its state to delta in order to journal in delta form upon the next commit operations. When the dirty block is in a delta state, it writes the dirty block in delta form to the NVRAM journal to reduce the access latency and space consumption of the NVRAM. After all dirty blocks are committed, DJFS writes the data of JC block to the NVRAM journal to further reduce the volume of the journal metadata.

As mentioned above, difference capturing and delta compression are required for Delta Journaling. For difference capturing, DJFS copies the original block in the buffer cache before making a modification. It maintains a list of the original blocks in DRAM to avoid reading them from file system storage. Then, during the commit phase, DJFS creates a difference block between the original and

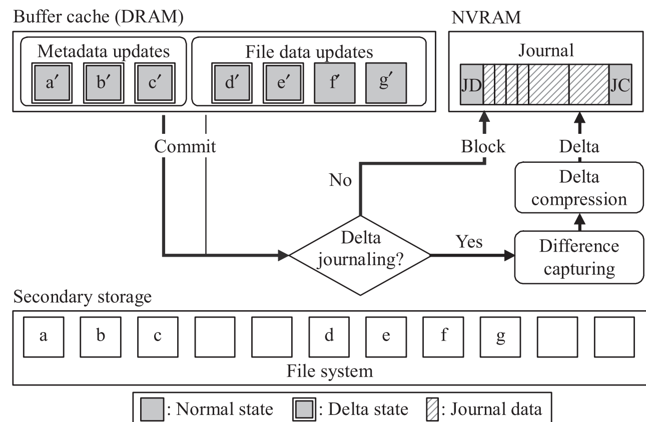

Fig. 4. Overall architecture of DJFS and the commit operation of DJFS.

dirty blocks using a bit-wise XOR operation on the two blocks. If multiple writes are issued to the same block in a single transaction, the process of difference capturing is required only once because the block in the buffer cache absorbs multiple writes during the time interval between commit operations. After capturing the difference, the copy of the original block is dropped for efficient use of DRAM space. For the delta compression, DJFS uses the LZO lossless compression algorithm [22], which was selected through experiments that are not described in the current paper owing to a space limitation. The difference block, which is made as a result of an XOR operation, is compressed and then written to the NVRAM journal. It is certain that as the size of the updates decreases, the results of the XOR are dominated by zeros, and thus the compression ratio can be high. The computation overhead of the difference capturing and delta compression is negligible when compared to block-level journaling.

3. Aggressive Group Checkpointing

The second approach used to relieve the overhead of explicit checkpointing is to exploit the idle time of the storage device. As mentioned previously, the checkpoint transactions are reflected to the file system opportunistically once they have been committed. Next, if the journal area does not have a sufficient amount of free space to handle upcoming write requests, explicit checkpointing is triggered to reflect the checkpoint transactions into the file system through disk cache flush operations and FUA commands, until the minimal necessary journal space is reclaimed. Because the default Ext4 journaling shares the file system region and journal area in a single storage device, there is no choice in the current checkpointing mechanism.

Compared to the original journaling, we do not issue write requests to the storage device upon periodic commit operations or frequent fsync() calls through the efficient use of small-sized NVRAM as the journal area. Thus, DJFS has the opportunity to exploit the idle time of the storage device. To relieve the overhead of explicit checkpointing, DJFS logically divides the NVARM journal space into *n* checkpointing regions. In the example shown in Fig. 5, we logically divide the NVRAM journal into two regions. When one region becomes full, DJFS simply switches to another region to handle upcoming write requests maintained by the newly running transaction. At this point, to guarantee a recovery from a sudden system failure, DJFS converts the state of dirty blocks in the checkpoint transactions into a normal state, and thus upcoming write requests will first be committed in the form of an entire block to another region.

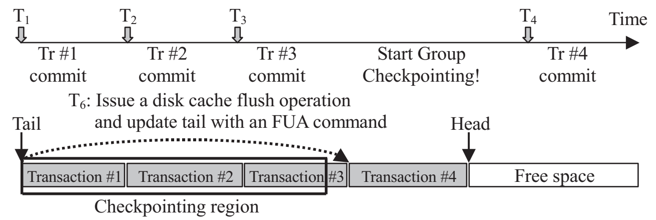

Fig. 5. Overall sequence of Aggressive Group Checkpointing.

Meanwhile, because the checkpointing will not interfere with upcoming journal writes, DJFS aggressively reflects the checkpoint transactions in the previous checkpointing region into the file system device using a disk cache flush operation and FUA command. Through this technique, DJFS reclaims a large amount of journal area with only a single disk cache flush operation and a single FUA command, and thus secures the free space of the NVRAM journal more quickly. Determining the number of checkpointing regions is related to the I/O pattern of the workloads and throughput of the file system device.

4. System Recovery

A sudden power loss or system failure can result in an inconsistent state of the file system. After an abnormal termination, DJFS restores the file system to a consistent state without performing time-consuming consistency checks on the entire file system. In the following, we discuss how DJFS utilizes the journal data of NVRAM for a recovery in detail. Figure 6 shows the workings of DJFS during the recovery phase. When a system crash occurs while committing a transaction, the committed transaction may be partially logged in the NVRAM journal area. In this case, to guarantee the atomicity of the transaction, we simply invalidate the partially logged data from the NVRAM journal area. In the example shown in Fig. 6,

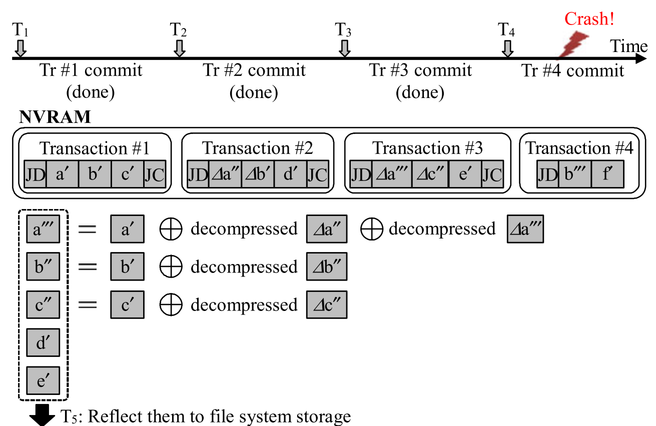

Fig. 6. DJFS operating in the recovery phase.

because a system crash occurs before completely logging the fourth transaction, DJFS invalidates the fourth transaction during the recovery phase. Consequently, DJFS can be restored to the last committed consistent state by reflecting all journal data of fully logged transactions into their original locations. As mentioned previously, to ensure a recovery from a system failure, DJFS first journals each dirty block in the form of an entire block to the NVRAM journal. Thus, the updated contents of blocks a′, b′, c′, d′, and e′ are first logged in the form of an entire block, and if multiple commit operations are issued to them, the dirty blocks are then logged in a delta form. When a dirty block is logged in delta form, the committed dirty block is made through an XOR operation between the previous modification and the decompressed delta, as shown in Fig. 6. Through this procedure, DJFS can replay all fully logged transactions while ensuring recovery to a consistent state, even when a system crash occurs during checkpointing or recovery.

VI. Evaluation

1. Experimental Setup

For the evaluation, we implemented DJFS based on Ext4 in the Linux kernel version 4.1, and then installed our prototype on a conventional machine equipped with an Intel Core i7-3770K CPU and 8 GB of DRAM. We used 120 GB Samsung 840 SSDs for the secondary storage devices. In addition, we allocated 128 MB of DRAM (equal to the size of the default journal area) as our target journal device, in order to simulate NV-DRAM or STT-MRAM. To correctly emulate the performance of NVRAM in the existence of volatile CPU caches, we log the journal data into NVRAM by using a non-temporal memory copy, which includes a cache line flush and memory fence operations (for example, clflush and mfence) [23].

All experiments were conducted using default journaling configurations. We first compared DJFS with ordered and full journal modes of Ext4 that adopt an on-disk journal (OJ-O and OJ-F, respectively). We also compared DJFS with Ext4 full journal mode, which adopts an external journal (EJ-F), in order to verify the effectiveness of removing journal write traffic from the file system storage. For the performance experiments of DJFS, we first examined the performance of our first optimization scheme, called Journaling without Flushes (NVJ), in order to analyze the performance of DJFS in more detail. We then examined the performance of DJFS using two and four checkpointing regions (NVJ + DJ + AGC2 and NVJ + DJ + AGC4).

2. Runtime Performance

We first evaluated the random write performance of DJFS using the FIO benchmark [13]. We conducted experiments using a random write pattern and 4K files, each of which has a size of 1 MB, by varying the frequency of fsync() calls from never to once every 64 writes, in order to evaluate the performance of DJFS for both system-wide and application-level consistency. Figure 7 shows the write IOPS of FIO benchmark. In the case of no fsync() mode, DJFS outperforms the on-disk journaling of Ext4 full journal mode by up to 1.7 times. Furthermore, its performance is close to the performance of the on-disk journaling of Ext4 ordered mode, although DJFS guarantees a higher level of system-wide consistency. This is because we eliminate journal write traffic from frequent disk cache flush operations and FUA commands on the file system device.

In addition, when fsync() calls were used in the experiments, DJFS outperformed both on-disk journaling of Ext4 ordered and full journal modes by up to 5.1 times. The reason for this is that, as the frequency of fsync() calls increases, the numbers of disk cache flush operations and FUA commands increase dramatically in the on-disk journaling of Ext4. Meanwhile, DJFS does not suffer from JOJ anomalies, despite fsync() calls being heavily used to ensure application-level consistency. When compared to the external journaling of Ext4, DJFS improves the write IOPS by up to 4.0 times. In addition, when compared to NVRAM journaling, DJFS improves the write IOPS by up to 1.8 times because the combination of our optimizations can relieve the explicit checkpointing overhead.

To evaluate the performance of DJFS for application-level consistency, we also used the TPC-C SQLite database benchmark [24]. The TPC-C workload produces a large number of small reads and writes. During the experiments, the number of warehouses was set to ten. We measured the transactions processed per minute (tpmC) in

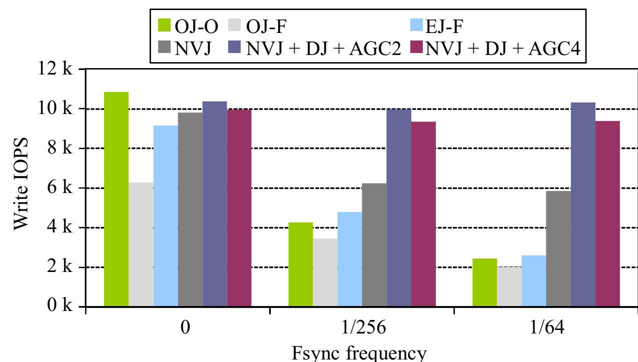

Fig. 7. Runtime performance of FIO.

both SQLite Rollback and WAL modes. For Rollback mode, we used the PERSIST mode, which is widely known as the fastest mode among all Rollback modes. Fig. 8 shows the tpmC of TPC-C SQLite benchmark. DJFS outperforms both on-disk and external journaling of Ext4 by up to 64.2 times and 26.5 times in SQLite Rollback and WAL modes, respectively. This is because SQLite frequently triggers fsync() calls to maintain the database transactions, and thus issues many disk cache flush operations and FUA commands, as shown in Table 1. In addition, such overhead is particularly severe in Rollback mode owing to the high frequency of fsync() calls [15]. Thus, both on-disk and external journaling of Ext4 suffer from a non-trivial delay. As the experimental results indicate, the average delta size per block is only 6.7% and 22.1% of the block size in SQLite Rollback and WAL modes, respectively. The results indicate that our second optimization scheme, Delta Journaling, greatly reduces the access latency and space requirements of the NVRAM journal, by efficiently exploiting the byte-addressable characteristics of NVRAM.

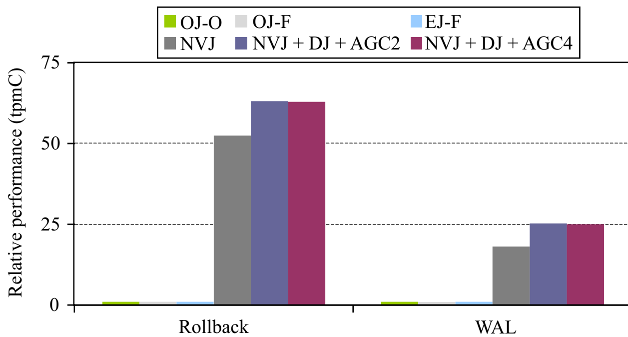

Fig. 8. Runtime performance of TPC-C SQLite.

Table 1. Disk cache control and journal writes of TPC-C SQLite (normalized by OJ-O).

		OJ-O	OJ-F	EF-F	NVJ	2W	4W
Rollback							
Disk cache control (%)	FLUSH	100	99.2	99.5	0.4	0.003	0.006
	FUA	100	99.3	99.6	0	0	0
Journal writes (%)	JM	100	99.3	99.5	104.6	2.1	2.1
	Data	100	297.4	298.2	239.8	33.7	35.0
WAL							
Disk cache control (%)	FLUSH	100	99.3	99.3	1.3	0.02	0.03
	FUA	100	100	100	0	0	0
Journal writes (%)	JM	100	100	100	103.7	2.6	2.7
	Data	100	313.3	312.6	274.6	93.4	101.1

*JM: journal metadata including journal superblock, JD, and JC.

3. Effects of NVRAM Write Latency

To cover a wide range of NVRAM devices, we examined the effects of NVRAM write latency on DJFS. We performed experiments using the TPC-C SQLite database benchmark, and measured the tpmC by varying the write latency of NVRAM from 0 to +40 ns, compared to that of DRAM. Assuming that the write latency of DRAM is 10 ns, the write latency of NVRAM is from 1 to 5 times slower. In the experiments, we used PERSIST as SQLite journaling mode.

Figure 9 shows the transaction throughput of DJFS by varying the write latency of an NVRAM device. The transaction throughput of NVRAM journaling and DJFS is normalized through external journaling of Ext4 full journal mode. As the write latency of NVRAM increases compared to that of DRAM, the transaction throughput of NVRAM journaling decreases rapidly. On the other hand, as the write latency of NVRAM increases compared to that of DRAM, the transaction throughput of DJFS slightly decreases when compared to that of NVRAM journaling; DJFS outperforms NVRAM journaling by up to 3.52 times. This is because we devoted significant effort to reduce the volume of journal data through the Delta Journaling scheme, and thus the write latency of the NVRAM device significantly decreases. In addition, even though the write latency of NVRAM increases compared to that of DRAM, both NVRAM journaling and DJFS outperform on-disk journaling of Ext4 full journal mode; even under the worst case situation, NVRAM journaling and DJFS outperform on-disk journaling of Ext4 full journal mode by 11.06 times and 38.89 times, respectively. This means that our first optimization

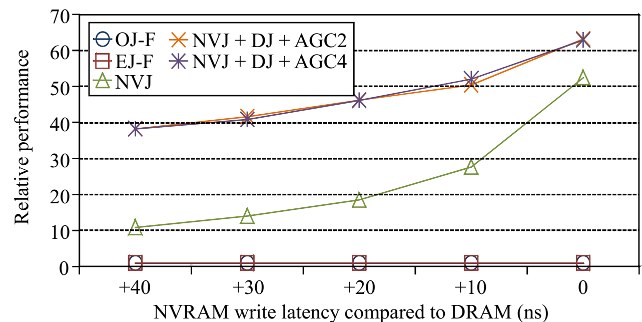

Fig. 9. Transaction throughput of DJFS by varying the write latency of NVRAM device compared to that of DRAM.

scheme, Journaling without Flushes, is quite effective for the application performance.

4. Buffer Cache Performance

Because DJFS copies the original block in the buffer cache before making a modification and maintains a list of original blocks in the DRAM for Delta Journaling, it may degrade the space efficiency of the buffer cache. To investigate the memory overhead when storing the original blocks, we compared the buffer cache miss ratio of the original buffer cache scheme, which uses the DRAM space only for caching (BF-Original), along with DJFS. During the experiments, we configured the size of DRAM from 10% to 100% relative to the I/O footprint. Thus, only cold misses occurred in the original buffer cache scheme when the size of the DRAM was 100%. As shown in Fig. 10, the buffer cache performance of DJFS exhibits only a marginal degradation when compared to that of the original buffer cache scheme under all I/O workloads. This is because we drop the copy of the original block after capturing the differences for an efficient use of the DRAM space. In addition, DJFS first journals each dirty block in the form of the entire block to the NVRAM journal, and thus only frequently modified blocks are involved in the copying process. The last reason for this is that I/O workloads have a non-uniform access frequency distribution [25]. Specifically, the update frequency of the I/O workloads is highly skewed, and thus the list of original blocks in DJFS occupies only a small portion of the DRAM space.

VII. Related Works

Consistency Overhead. Some studies have focused on a JOJ anomaly, which originates from the fact that the file system journals the database journaling activity. Using optimal journaling mode in SQLite, adopting external journaling, and exploiting polling-based I/O, Jeong and others [15] improved the smartphone performance by optimizing the Android I/O stack in such a way that unnecessary metadata journaling is eliminated for the file system. Min and others [6] presented a new file system, CFS, which supports a lightweight native interface for applications to maintain crash consistency of the application data on transactional flash storage. To guarantee application consistency, CFS uses three key techniques: the selective atomic propagation of dirty pages, in-memory metadata logging, and delayed allocation. Our work differs from these studies in that we directly exploit file system-level semantics only, and thus do not require any modification of the application, device driver, or storage firmware.

Other studies have focused on the journaling overhead in various storage architectures. Lee et al. [26] proposed a new buffer cache architecture, UBJ, which eliminates the storage accesses upon a commit operation by using an in-place commit technique. With this technique, the caching and journaling functions are subsumed in a unified NVRAM space. However, UBJ requires a sufficiently large NVRAM to cover them. In contrast to this work, we can help reduce the cost of expensive NVRAM, while satisfying the desired level of performance. In a recent study [27], DuraSSD was shown to eliminate the need for write barriers with disk cache flush operations by making the DRAM write-cache inside durable SSD using tantalum capacitors. In addition, some high-end SSDs contain a power-protected DRAM write-cache. DJFS further improves the performance and lifespan of these products by reducing the write traffic to them, although DJFS was designed for broader use.

NVRAM Utilization: Some studies have focused on an NVRAM/NAND flash hybrid architecture that utilizes the byte-accessibility and in-place updating of NVRAM to complement the hardware limitations of NAND flash-based storage. First, Sun et al. [28] utilized NVRAM as a log region inside NAND flash-based storage. Thus, small updates can be effectively absorbed in this hybrid storage device because the NVRAM log region allows in-place updates. The second approach utilized NVRAM as a metadata storage component [20]. The reason for this is that file system metadata are frequently updated, and only a few bytes are actually modified. Thus, to reduce excessive garbage collection overhead in NAND flash-based storage, they utilized NVRAM to log the file system metadata.

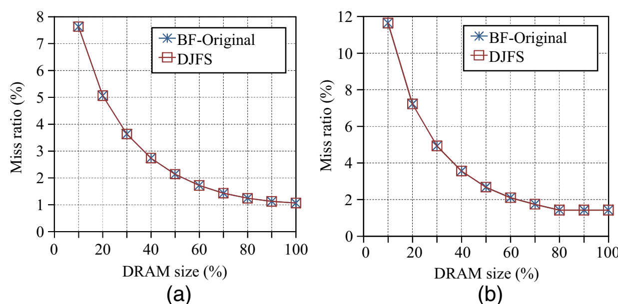

Fig. 10. Miss ratio of original buffer cache scheme and our file system: (a) SQLite Rollback and (b) SQLite WAL.

VIII. Conclusion

In this paper, we proposed DJFS that efficiently utilizes small-sized NVRAM for a file system journal in order to

provide both a high level of performance and data consistency for different applications. To achieve our goal, we made three technical contributions: Journaling without Flushes, Delta Journaling, and Aggressive Group Checkpointing. Our experiment results clearly show that DJFS outperforms Ext4 by up to 64.2 times with only 128 MB of NVRAM. This work will contribute to an analysis of the overhead for ensuring data consistency. In addition, we presented a practical file system solution that does not require any modifications to the applications, device drivers, or storage firmware, and incurs only a slight increase in hardware costs. In addition, our solution can be easily applied to many different types of systems.

Acknowledgements

This work was supported by MSIT (Ministry of Science and ICT), Rep. of Korea, under the SW Starlab support program(IITP-2015-0-00284) supervised by the IITP (Institute for Information & communications Technology Promotion) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2015M3C4A7065696).

References

[1] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-Tree Filesystem,” *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013, pp. 1–32.

[2] M. Rosenblum and J.K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” *ACM Trans. Comput. Syst.*, vol. 10, no. 1, Feb. 1992, pp. 26–52.

[3] A. Sweeney et al., “Scalability in the XFS File System,” *USENIX Annu. Techn. Conf. (ATC)*, San Diego, CA, USA, Jan. 22–26, 1996, pp. 1–14.

[4] V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, “Analysis and Evolution of Journaling File Systems,” *USENIX Annu. Techn. Conf. (ATC)*, Anaheim, CA, USA, Apr. 10–15, 2005. pp. 105–120.

[5] SQLite Team, SQLite, SQLite, Accessed Oct. 26, 2015. http://www.sqlite.org/

[6] C. Min et al., “Lightweight Application-Level Crash Consistency on Transactional Flash Storage,” *USENIX Annu. Techn. Conf. (ATC)*, Santa Clara, CA, USA, July 8–10, 2015, pp. 221–234.

[7] H. Kim, N. Agrawal, and C. Ungureanu, “Revisiting Storage for Smartphones,” *USENIX Conf. File Storage Technol.*, San Jose, CA, USA, Feb. 14–17, 2012, pp. 209–222.

[8] Apple Corporation, BSD System Calls Manual: FCNTL(2), *Apple*, 2001. Accessed Feb 11, 2015. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fcntl.2.html

[9] Y. Zhang et al., “ViewBox: Integrating Local File Systems with Cloud Storage Services,” *USENIX Conf. File Storage Technol.*, Santa Clara, CA, USA, Feb. 17–20, 2014, pp. 119–132.

[10] S.C. Tweedie et al., “Journaling the Linux Ext2 fs Filesystem,” *Annu. Linux Expo*, Durham, USA, May 28–30, 1998, pp. 1–8.

[11] A. Aghayev et al., “Evolving Ext4 for Shingled Disks,” *USENIX Conf. File and Storage Tech. (FAST)*, Santa Clara, USA, Feb. 27–Mar. 2, 2017, pp. 105–119.

[12] W. Lee et al., “WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly,” *USENIX Annu. Techn. Conf.*, Santa Clara, CA, USA, July 2015, pp. 235–247.

[13] J. Axboe, FIO (Flexible IO Tester), 2005, Accessed June 15, 2014. http://freecode.com/projects/fio/

[14] W.-H. Kang et al., “X-FTL: Transactional FTL for SQLite Databases,” *ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, USA, June 22–27, 2013, pp. 97–108.

[15] S. Jeong et al., “I/O Stack Optimization for Smartphones,” *USENIX Annu. Techn. Conf.*, San Jose, CA, USA, June 26–28, 2013, pp. 309–320.

[16] B. Lee et al., “Architecting Phase Change Memory as a Scalable DRAM Alternative,” *Int. Symp. Comput. Archit.*, Austin, TX, USA, June 20–24, 2009, pp. 2–13.

[17] J. Condit et al., “Better I/O Through Byte-Addressable, Persistent Memory,” *ACM SIGOPS Symp. Operating Syst.*, Principles, Big Sky, MT, USA, Oct. 11–14, 2009, pp. 133–146.

[18] X. Wu and A.L. Reddy, “SCMFS: A File System for Storage Class Memory,” *Int. Conf. High Performance Comput., Netw., Storage Anal.*, Seattle, WA. USA, Nov. 12–18, 2011, pp. 1–11.

[19] X. Guo, E. Ipek, and T. Soyata, “Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing,” *Int. Symp. Comput. Archit.*, Saint-Malo, France, June 19–23, 2010, pp. 371–382.

[20] C. Lee and S.-H. Lim, “Efficient Logging of Metadata Using NVRAM for NAND Flash based File System,” *IEEE Trans. Consumer Electron.*, vol. 58, no. 1, Feb. 2012, pp. 86–94.

[21] K. Kim et al., “IPL-P: In-Page Logging with PCRAM,” *VLDB Endowment*, vol. 4, no. 12, Aug. 2011, pp. 1363–1366.

[22] M. Oberhumer, LZO Real-Time Data Compression Library, *GmbH*, 1996. Accessed Aug. 13, 2015. http://www.oberhumer.com/opensource/lzo/

[23] F. Chen, M.P. Mesnier, and S. Hahn, “A Protected Block Device for Persistent Memory,” *Int. Conf. Massive Storage Syst. Technol.*, Santa Clara, CA, USA, June 2–6, 2014, pp. 1–12.

[24] S.T. Leutenegger and D. Dias, “A Modeling Study of the TPC-C Benchmark,” *ACM SIGMOD Int. Conf. Manage. Data*, Washington, DC, USA, May 26–28, 1993, pp. 22–31.

[25] D. Roselli, J.R. Lorch, and T.E. Anderson, “A Comparison of File System Workloads,” *USENIX Annu. Techn. Conf.*, San Diego, CA, USA, June 18–23, 2000, pp. 41–54.

[26] E. Lee, H. Bahn, and S.H. Noh, “Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory,” *USENIX Conf. File Storage Technol.*, San Jose, CA, USA, Feb. 12–15, 2013, pp. 73–80.

[27] W.-H. Kang et al., “Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases,” *ACM SIGMOD Int. Conf. Manage. Data*, Snowbird, UT, USA, June 22–27, 2014, pp. 529–540.

[28] G. Sun et al., “A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement,” *IEEE Int. Symp. High Performance Comput. Archit.*, Bangalore, India, Jan. 9–14, 2010, pp. 51–77.

Junghoon Kim received his BS degree in the Department of Computer Engineering, MS degree in the Department of Mobile Systems Engineering, and PhD degree in the Department of IT Convergence from Sungkyunkwan University, Rep. of Korea, in 2010, 2012, and 2016, respectively. Since 2016, he has been a senior software engineer at Samsung Electronics, Rep. of Korea. His research interests include storage systems, embedded systems, virtualization, and operating systems.

Minho Lee received his BS degree in the Department of Electronic and Electrical Engineering, MS degree in the Department of Electrical and Computer Engineering from Sungkyunkwan University, Suwon, Rep. of Korea, in 2013 and 2015, respectively. He is currently a PhD candidate in the Department of Electrical and Computer Engineering at Sungkyunkwan University. His research interests include operating systems, virtualization, and file systems.

Yongju Song received his BS degree in Computer Engineering from Korea Polytechnic University, Siheung, Rep. Korea, in 2015. He is currently working toward a combined master's and doctorate program in the Department of Electrical and Computer Engineering at Sungkyunkwan University, Suwon, Rep. of Korea. His research interests include storage systems, operating systems, and file systems.

Young Ik Eom received his BS, MS, and PhD degrees in Computer Science and Statistics from Seoul National University, Rep. of Korea, in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an associate professor at Dankook University, Seoul, Rep. of Korea. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine, USA, from Sept. 2000 to Aug. 2001. Since 1993, he has been a professor at Sungkyunkwan University, Suwon, Rep. of Korea. His research interests include virtualization, file systems, operating systems, and cloud systems.