S-WAL: Fast and Efficient Write-Ahead Logging for Mobile Devices

Dong Hyun Kang, Woonhak Kang, and Young Ik Eom

***Abstract*—A crash-consistency mechanism of database application (hereinafter DBMS for short) imposes an enormous burden on the journaling process of the file system (e.g., JBD2) in that it employs regular files on the file system to persistently preserve the application's journal data with synchronous system calls. Unfortunately, much of the previous research is impractical in real-world scenarios because they require changing a lot of source lines of code in multiple software layers of the software stack or using special hardware (e.g., the transactional flash storage or NVM technologies). In this paper, we propose a new journal mode of DBMS, called S-WAL, which compresses the raw data journaled by the database to alleviate both the redundant journaling operations and harmful write amplification. We believe that S-WAL is a practical way to support application-level crash consistency on the existing mobile devices because only a few lines of DBMS code need to be changed, without the need to employ special hardware. We demonstrate the effectiveness of S-WAL by running four popular mobile applications on the latest smartphone. Our evaluation results show that S-WAL considerably outperforms existing journal modes of DBMS in all cases. In the best case, S-WAL reduces the elapsed time by up to 7.5× more than the PERSIST mode and by up to 51% more than the traditional WAL mode.**

***Index Terms*—Mobile applications, write-ahead logging, database systems.**

I. Introduction

MOBILE devices such as smartphones and tablets have become a common part of our lives and are now being used to process sensitive data (e.g., email, banking, shopping, and social network services). Therefore, guaranteeing the data consistency and durability becomes one of the most critical issues in using mobile devices [1]–[4]. Fortunately, database application (hereinafter DBMS for short) implements its own update protocols (e.g., journaling and write-ahead logging) to efficiently achieve *application-level crash consistency*; it is therefore employed by most mobile applications as their database engine [5]. However, the crash-consistency mechanism of DBMS imposes an enormous burden on the journaling process of the file system (e.g., JBD2) because it employs regular files on the file system to persistently preserve the application's journal data with synchronous system calls (e.g., fsync() and fdatasync()) [2]. In addition, DBMS must align all I/O requests at page granularity of the kernel (e.g., 4KB or 8KB), even though only a small number of bytes are modified by a single query [6]–[8]. For example, assume that a message application sends a single insert query with two characters, "Hi". In this case, DBMS must submit 4KB of writes to the kernel for the page alignment. Therefore, DBMS suffers from a journaling of journal (JOJ) [2] and write-amplification factor (WAF) problem [6].

A variety of studies have been performed to solve the above practical issues [2]–[4], [6], [9]–[17]. For example, to solve the JOJ problem, some research groups have proposed new interfaces over the I/O stacks [2], [4], [11], [13]–[15], while other research groups focused on the database engine of DBMS with special hardware (e.g., transactional flash storage) [12], [17]. Recently, Oh *et al.* [6] and Kim *et al.* [16] proposed delta-based consistency mechanisms with Non-Volatile Memory (NVM) to address the WAF problem of DBMS. Unfortunately, much of the previous research is impractical in real-world scenarios because the proposed schemes require many of the source lines of code to be changed at multiple software layers of the software stack. For example, to support legacy applications, CFS [12], File-Adaptive Journaling [13], and X-FTL [17] must modify not only the engine of DBMS, but also kernel code. In CFS, about 5,800 lines of code are changed to solve the consistency problem. In addition, NVM technologies are not yet available in the mobile market and we envision that deploying NVM in mobile devices is quite difficult since the manufacturing density would be insufficient and the cost too high to embed them into mobile devices.

In this paper, we propose a new journal mode, called S-WAL, which compresses the raw data journaled by DBMS to mitigate both the redundant journaling operations and the harmful write amplification. We believe that S-WAL is a practical way to support *application-level crash consistency* on the existing mobile devices because only a few lines of the DBMS code need to be changed, without the need to employ special hardware such as transactional flash storage or NVM technologies. We make the following technical contributions:

Analysis of Journal Mode and Data: We first studied the existing journal modes of DBMS with the I/O behavior of each journal mode, and then analyzed both the performance of each

Manuscript received June 13, 2018; accepted July 16, 2018. Date of publication July 25, 2018; date of current version September 24, 2018. This work was supported in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT under Grant NRF-2015M3C4A7065696, and in part by the Ministry of Science and ICT, Korea through the SW Starlab Support Program under Grant IITP-2015-0-00284 supervised by the Institute for Information and Communications Technology Promotion. *(Corresponding author: Young Ik Eom.)*

D. H. Kang and Y. I. Eom are with the College of Software, Sungkyunkwan University, Suwon 2066, South Korea (e-mail: kkangsu@skku.edu; yieom@skku.edu).

W. Kang is with Core Platform and Infrastructure Architecture, eBay, San Jose, CA, USA (e-mail: wokang@ebay.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCE.2018.2859630

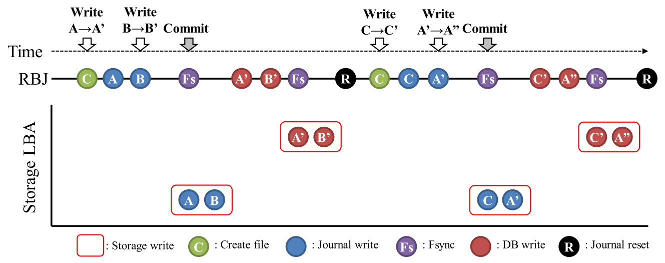

Fig. 1. Process of RBJ with storage I/Os.

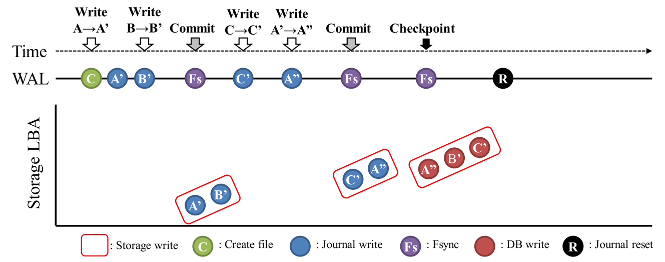

Fig. 2. Process of WAL with storage I/Os.

journal mode and the features of the raw data journaled by DBMS. We made two major observations from the analyses. First, the performance of WAL mode is comparable to that of MEMORY mode, even though WAL mode preserves both its database and WAL file persistently in the mobile storage. Second, a 4KB journal write of the DBMS can be compressed into 400B without the bitwise exclusive-or (XOR) operation.

Practical Design: We also studied the existing solutions that were proposed for mobile devices and compared S-WAL with the eight representative solutions in terms of practicability. We found that the S-WAL mode is practical and generic for two reasons. First, S-WAL is designed to avoid any modifications of not only kernel but also applications, and it requires only about 40 lines of code change on the code of the traditional WAL mode of DBMS. Therefore, mobile applications that link to the DBMS application's library for their database engine can easily change their journal mode on DBMS to S-WAL with a single query. Second, since hardware support is never required by S-WAL, it provides strong and complete backward-compatibility with the existing mobile devices.

Byte-Granularity Compression Logging: The key design challenge is to determine how to alleviate the massive performance overhead caused by the inherent mechanism of DBMS. Note that each single journal write of DBMS causes triggering a commit() operation on the file system because the write operation updates not only the corresponding file contents but also the associated metadata (i.e., inode) through fsync() or fdatasync() system call. To address this, we introduced a compression logging technique that compresses each journaled raw data to delay the journal write until a commit() operation of the DBMS is triggered.

SQL-Based Evaluation Framework: We also proposed a novel evaluation framework, called SQL-based evaluation framework, that helps present an in-depth study on the storage layer and measure the real performance of mobile devices. We implemented S-WAL on the framework on the target phone and conducted intensive experiments and measurements in terms of overall performance, space saving, compression cost, and recovery time. Our evaluation results show that S-WAL considerably outperforms existing journal modes of DBMS in all cases. In the best case, S-WAL reduces the elapsed time by up to 7.5× over the PERSIST mode and by up to 51% over the traditional WAL mode.

The content of the remaining paper includes a discussion of the background and design motivation in Section II, and a description of the details of S-WAL in Section III. We then present our evaluation framework and evaluation results in Section IV. Finally, we discuss the related work in Section V and conclude this paper in Section VI.

II. Background and Motivation

A. Database Application (DBMS)

Database application for mobile device (DBMS) has become one of the most popular database engines in mobile devices such as smartphones and tablets because almost all mobile applications (e.g., SNS, Mail, Browser, Game, and Streaming Services) use DBMS to maintain their data consistently. DBMS employs regular files on the file system to preserve both databases and journals of applications, and it fully guarantees both the data consistency and the durability under all failures by providing internal journal mechanisms: roll-back journal (RBJ) and write-ahead logging (WAL) [5], [18]. The five RBJ modes are: OFF, MEMORY, DELETE, TRUNCATE, and PERSIST. The OFF and MEMORY modes do not guarantee data consistency and durability. This is because the OFF mode completely disables journaling behaviors and the MEMORY mode maintains the journal file in memory (e.g., DRAM) to achieve high performance of applications. On the other hand, the other three journal modes, DELETE, TRUNCATE, and PERSIST mode, are widely used because they keep the journal file in the mobile storage (e.g., eMMC and SDcard) to ensure the data consistency and durability, where the DELETE mode is the default transaction mode in DBMS. However, compared to the OFF or MEMORY mode, the other three journal modes, DELETE, TRUNCATE, and PERSIST, usually have massive performance overhead, because they frequently issue I/O requests to the mobile storage to preserve a consistent state for the database.

The transaction processing of the RBJ modes involves three steps: 1) journal write step, 2) database write step, and 3) journal reset step. In order to ensure the atomicity of a transaction, DBMS utilizes synchronization system calls (e.g., fsync() and fdatasync()) after the completion of each step. Figure 1 depicts the process of the RBJ with its storage I/O patterns. In the first step, a journal file is created if a new transaction begins without a journal file. Then, the old content of database pages is recorded in the journal file with an update on the journal header for an UNDO transaction. In the second step, the new content is written to the database file (i.e., in-place update). Finally, the journal file is reset for reuse or is deleted according to the RBJ mode.

The DELETE mode always deletes the journal file when a transaction ends. Therefore, the performance of this mode is the worst among the RBJ modes because it issues a large number of storage I/O requests (e.g., inode and data block) to create a new journal file consistently whenever a transaction begins. To mitigate the negative effects of file creation,

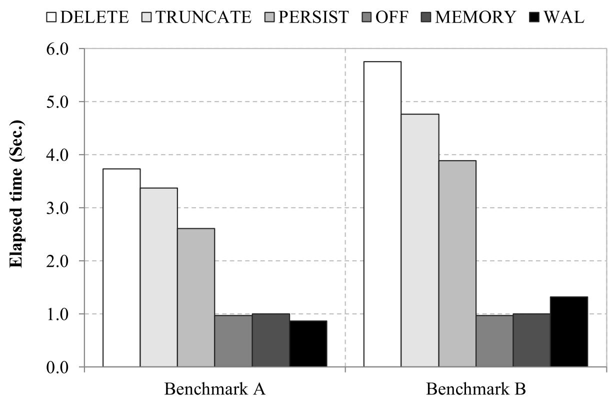

Fig. 3. The performance of each journal mode in DBMS.

the TRUNCATE mode only truncates the last journal file in the journal reset step and reuses it for the next transaction. Although this mode can reduce the number of storage I/O requests, it requires allocation of new data blocks to write the journal contents. On the other hand, the PERSIST mode simply changes the content of the last journal file to zero when a transaction is completed, and then uses the data blocks again without new block allocations. Therefore, the PERSIST mode is much more efficient than both the DELETE and TRUNCATE mode.

In contrast to the RBJ modes, the WAL mode maintains the original content in the database (Figure 2) and appends new content of the updated page to the WAL file as a separate log for a REDO transaction. The logs in the WAL file are reflected in the database whenever a checkpoint is triggered by a predefined threshold value for enabling automatic checkpoint or a close operation of the WAL file. Especially, this mode has three performance advantages over the RBJ modes. First, since the WAL mode can delay the update of database until the checkpoint is triggered, it reduces the number of storage I/O requests. Second, it can eliminate I/O requests issued to the database file because a separate log in the WAL file is invalidated when the corresponding page is rewritten, which is called as *page invalidation*. Finally, since the WAL mode sequentially writes the WAL file to append each log, it improves the performance of mobile storage; sequential pattern is more efficient than random one in mobile storage (called *flash-friendly write*).

B. Analysis of Journaling Performance

In the previous section, we described the internal journal mechanisms of DBMS. Now, more comprehensive study of each journal mode is still required in terms of the performance of mobile devices. We therefore performed a sensitivity analyses on the journaling performance. For fair comparisons, we employ the *SQL-based evaluation framework* that compares all journal modes of DBMS with the same set of traces (further details are described in Section IV-A). We first measure the performance of each journal mode with two popular mobile benchmarks [19], [20] on the target phone (Figure 3). In Figure 3, DBMS clearly reveals the performance impact of each journal mode. In particular, RBJ modes, which store both the database and journal

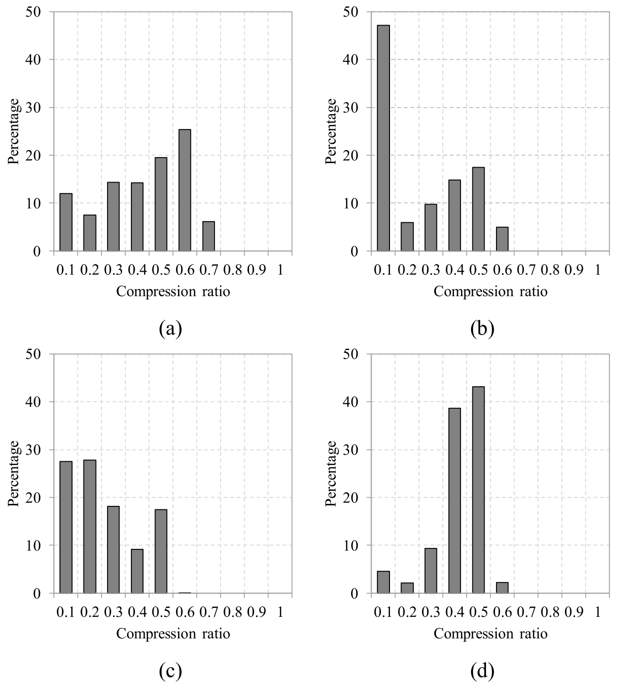

Fig. 4. Compression effect of raw data journaled by DBMS.

file in mobile storage, show poor performance because they lead to costly performance overheads, as mentioned in Section II-A. However, the performance of the WAL mode is comparable to that of the MEMORY mode. This is surprising because the WAL mode also maintains its database and WAL file in the mobile storage. To understand our performance results, we analyzed the SQL queries of the two benchmarks with I/O requests issued to the underlying storage and found out that most logs in the WAL file are not issued to the database file when the checkpoint is triggered because they are invalidated (i.e., *page invalidation*). In other words, it says that the amount of I/O requests that update its database file are significantly reduced during the execution. In addition, as mentioned in Section II-A, the WAL mode helps improve the performance of mobile storage by performing the *flash-friendly write*. Our experimental results clearly indicate that the journal mode in DBMS is correlated with the performance of the mobile device and strongly demonstrates why we exploit the traditional WAL mode instead of the RBJ mode in DBMS.

C. Compression Efficiency of DBMS Journal Data

In most mobile applications, the contents are partially updated on a database via the DBMS engine in order to guarantee the application-level crash consistency of the updates [6]–[8]. Unfortunately, this partial update negatively impacts mobile devices in terms of both performance and lifetime of the mobile storage because it can be considerably amplified while passing through the file system layer. In particular, the negative effects become more serious when the inserted queries are very small (e.g., create or drop the table of database). This characteristic of mobile data raises a question: can the harmful amplification be mitigated on the existing mobile device without any hardware support? To answer this

TABLE I
COMPARISON OF PROPOSED S-WAL WITH EIGHT REPRESENTATIVE SOLUTIONS PREVIOUSLY DESIGNED TO IMPROVE THE OVERALL PERFORMANCE

Solution name	Compression	Lines of code	Special hardware	Implementation	Compatibility*	Practicability**
S-WAL	Raw data	40 LoC	None	DBMS	Good	Good
WALDIO [4]	None	Not published	Reliable write	DBMS	Bad	Bad
LS-MVBT [11]	None	Not published	None	DBMS	Good	Bad
DBMS/PPL [6]	Delta	Not published	NVM technology	DBMS	Bad	Bad
NVWAL [16]	Delta	Not published	NVM technology	DBMS/Kernel	Bad	Bad
X-FTL [17]	None	Not published	Trans. storage	DBMS/Kernel	Bad	Bad
CFS [12]	None	< 5,800 LoC	Trans. storage	DBMS/Kernel	Bad	Bad
Adaptive jour. [13]	None	< 20 LoC	None	DBMS/Kernel	Good	Bad
MSYNC [14], [15]	None	< 200 LoC	None	Kernel/App.	Good	Bad

* The grade of compatibility is determined whether or not the solution requires special hardware.
** The practicability is defined based on both Lines of code ("Not published" means bad) and Implementation (code change of multiple layers means bad).

question, we compressed the journaled raw data (4KB granularity) while running popular mobile applications on a real mobile device (more details described in Section IV-A), to confirm whether the journal data compression has a positive performance effect or not. Figure 4 shows the distribution of the compression ratio, which is defined as the ratio of the size after compression to that before compression (i.e., a small compression ratio implies good compressibility). As shown in Figure 4, our experimental results confirm that the journal data compression is sufficient to resolve the above two problems: interestingly, all applications show smaller compression ratios, even though journaled raw data of the application is compressed without the bitwise exclusive-or (XOR) operation (i.e., delta). In particular, the results on Messenger show that about 50% of journal writes (4KB granularity) are transferred to approximately 400B (i.e., about 50% of journal writes have the compression ratio of 0.1). To better understand good compressibility, we also analyzed the content of each query in detail. We found that most update queries (e.g., insert, update, and delete) of Messenger are composed of very small messages, such as “Hi” and “What’s up?”. On the other hand, the largest compression ratio was observed in SNS, which is not surprising because SNS has a considerable amount of binary large object (BLOB) data. Even in this situation, more than 80% of the journal writes are compressed to around 2048B.

These observations further motivate us to propose the S-WAL mode because we believe that it is possible to resolve the massive performance overhead incurred by the inherent journaling mechanism of DBMS while guaranteeing the *application-level crash consistency*.

III. S-WAL: DESIGN AND IMPLEMENTATION

In this section, we first briefly present a comparison of the proposed S-WAL with the existing solutions to demonstrate its unique benefits. We then discuss the method of enhancing the traditional WAL mode to resolve the problem of redundant journaling and harmful write amplification.

A. Comparison Between S-WAL and the Existing Solutions

S-WAL has some unique characteristics compared to the previous approaches that had been proposed to improve the performance of mobile devices. Table I compares the characteristics of S-WAL with the existing solutions.

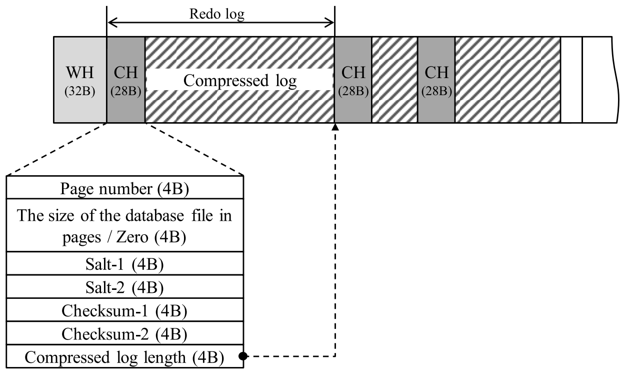

Fig. 5. The WAL file layout of S-WAL.

Raw Data Compression: To reduce the overhead of DBMS journaling, most previous solutions adopted delta-based compression, which compresses the difference between the original and changed data (i.e., delta) [6], [9], [10], [16]. However, in the delta-based approach, the raw data on the original location must be persistently copied into another location because the delta can be partially reflected to the data on the original location by unexpected system crashes. Unfortunately, the delta-based approach also suffers from the performance penalty caused by an amplified read request since the delta-based approach needs to read the original data and all consecutive deltas after a commit() operation to service any small-size read request. On the other hand, S-WAL does not have such overheads; instead, S-WAL itself compresses the journaled raw data. To the best of our knowledge, this work is the first study in which the journaled raw data is compressed instead of using the delta.

Practical Design: To guarantee the application-level crash consistency, many researchers have proposed a new ioctl() interface by modifying the code of the kernel [12]–[15], [17]. Therefore, to achieve crash consistency, developers need to define their ioctl() interface in their applications or in the DBMS engine code (i.e., source lines of code need to be modified across multiple software layers). Meanwhile, researchers have focused on the DBMS engine and modified the code of DBMS to improve the performance of the mobile device while guaranteeing the crash consistency [2], [4], [6], [11], [16].

However, the proposed solutions are not intuitive in terms of implementation and seem to require a lot of changes in the source lines of code. In particular, since the header embedding technique of WALDIO [4] may decrease the compatibility with the existing journal modes of DBMS, adopting the scheme in real-world devices (or systems) is impractical. On the other hand, S-WAL is intuitively designed and only requires about 40 lines of DBMS code modifications.

No Special Hardware Support: Some proposed solutions employ special hardware features, such as transactional flash storage [12], [17], NVM technologies [6], [10], [16], [21], and reliable write command of eMMC [4]. Unfortunately, these features are not yet common in actual mobile devices. On the other hand, S-WAL does not require any special hardware support. Consequently, S-WAL is more practical and general than the existing proposed solutions in terms of backward compatibility.

B. Byte-Granularity Compression Logging

S-WAL compresses the raw data journaled by DBMS, where we call it compressed log, and temporarily buffers it into the DRAM memory to overcome the high storage I/O cost imposed by DBMS by replacing storage I/O with the less costly CPU and memory copy. Such an exchange is possible because mobile CPU and DRAM technologies have rapidly evolved, promising a lower overhead than that of storage I/O (e.g., redundant journaling operation and page granularity write).

1) File Layout of S-WAL: In traditional DBMS, a WAL file consists of a set of redo logs and each redo log contains a 24B frame header and a 4KB WAL frame [22]. The frame header contains a page number to which the associated WAL frame should be reflected, and each page number is used to reconstruct the WAL index after system crash. In order to follow this format with the compression, we re-designed the layout of the WAL file so that each redo log is composed of a 28B compression header and a variable-length *compressed log*. Figure 5 depicts the modified file layout of S-WAL. The modified WAL file is composed of a WAL header (WH) and a set of redo logs. The WAL header includes basic information of the WAL file, such as magic number, file format version, database page size, etc. Next, the redo log is a unit of handling a redo transaction and it consists of a compression header (CH) and a compressed log. The compression header maintains information about the subsequent compressed log, such as page number, salt, checksum, and the compressed log length. In S-WAL, the compressed log length helps to identify the end of the compressed log because the length of each compressed raw data can be highly variable and depends on the compression ratio of the raw data.

2) Single Journal Write and Commit: DBMS is designed to work with its own in-memory buffer in the user space to batch write operations. Once a commit() operation is triggered, the updated dirty pages on the buffer are flushed one by one to the corresponding WAL file in redo log form. The core design goal of S-WAL is to group multiple journal writes for redo logs into a single journal write. To achieve the design goal, S-WAL compresses each journal data on the buffer and appends it to

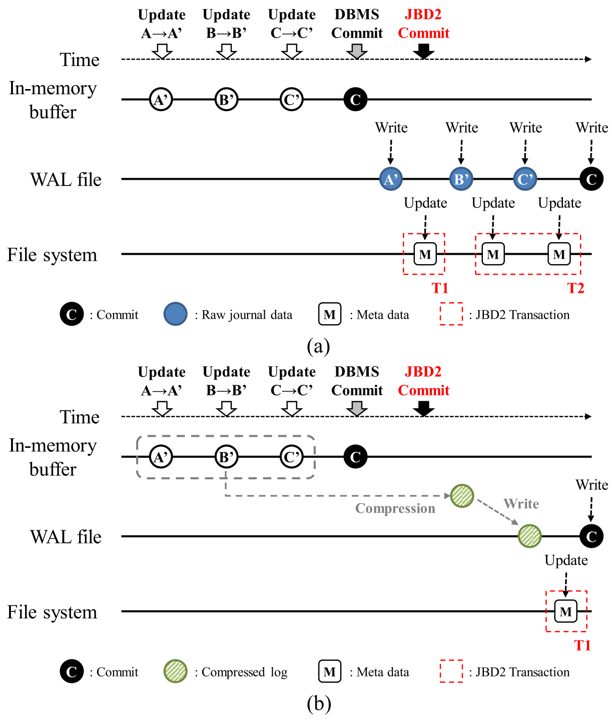

Fig. 6. Running example of S-WAL compared to that of WAL mode.

the auxiliary in-memory buffer, where we call it *compressed buffer*, instead of writing each of journal data to the WAL file on the mobile storage. The delayed compressed logs are flushed with compression headers to the WAL file immediately before the commit() operation. Therefore, S-WAL can defer a journal write operation of DBMS as much as possible. Fortunately, such a delay does not affect the crash consistency because the journal data is considered durable when the commit() operation has been safely completed.

Figure 6 illustrates a running example of S-WAL compared to that of the WAL mode. For example, assume that a messenger application triggers a commit() operation after updating three pages with very small queries. In this example, the traditional WAL mode separately performs four journal writes (three for the updated pages and one for the commit page) to ensure their durability, because the WAL mode issues the updated dirty pages on the in-memory buffer one by one to the WAL file. Unfortunately, this mode might impose an unintended burden on the journaling daemon of the file system (e.g., JBD2) because the metadata for the WAL file is always updated when each write operation is completed on the file system. In the S-WAL mode, the three journal writes can be combined into a single journal write by compressing them on-the-fly before the commit page is written to the WAL file. Consequently, S-WAL issues only two journal writes (one for the page that contains a set of compressed logs and the other for the commit page). In addition, the metadata of the WAL file might not be journaled by the file system by narrowing the range of consecutive writes. This idea is simple and straightforward compared to the existing solutions, and renders S-WAL faster and more efficient.

3) Abort and Checkpoint: If an abort() operation is triggered by an application before a checkpoint, S-WAL drops

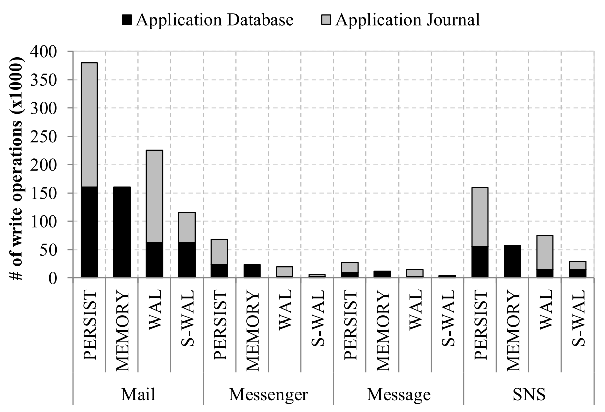

Fig. 7. Number of storage writes caused by DBMS.

all up-to-date pages on DBMS engine and the compressed logs associated with the aborted transaction will be ignored in the future recovery procedure. Note that the database persistently stores the original data. As mentioned in Section II-A, a checkpoint procedure is periodically triggered to update the original data on a database in batch way. When a checkpoint() operation occurs, S-WAL follows the basic rules of the WAL mode except that the up-to-date data stored in compressed form is reflected in the original form with the decompression.

4) Recovery: The recovery procedure of S-WAL does not significantly differ from that of the traditional WAL mode except for the decompression phase. When the DBMS engine restarts after a crash or failure, S-WAL detects the data inconsistency by simply verifying the WAL file, which preserves crash recovery information including the checkpoint sequence number, abort records, and commit records. If the last record in the WAL file is not a commit record, S-WAL triggers the recovery procedure on-the-fly. This is possible because S-WAL always leaves a commit record after finishing each transaction. Upon a recovery, S-WAL reads the successfully committed compressed logs to decompress them one by one. The decompressed data is reflected in the original database by the next checkpoint operation.

5) Implementation: To implement S-WAL in the DBMS engine (version 3.8.6), we modified about 40 lines of code. We also employed a common compression library in mobile environments as the compression algorithm of S-WAL. We added 15 LoC in the main function to compress the raw data journaled by DBMS. A dozen lines of code is inserted immediately before the recovery and checkpoint operation for decompressing compressed logs.

IV. Evaluation

In this section, we propose a novel evaluation framework to help present an in-depth study of the storage layer and measure the real performance of mobile devices. We then present the results of the performance experiments in response to the following questions:

- What are the performance gains of S-WAL mode?
- How does S-WAL mode impact on the performance?
- What is the performance overhead of S-WAL at runtime?

TABLE II
Specification of Target Phone

	Specifications
CPU	2.7GHz quad-core CPU
Memory	SDRAM 3GB
Storage	eMMC 32GB
Kernel version	Mobile OS ver. 5.0.0_r3.0.1
DBMS version	DBMS 3.8.6
File system	Ext4 with JBD2 (Ordered mode)

A. SQL-Based Evaluation Framework

We first describe a novel evaluation framework in detail. The design goal of our evaluation framework is to study not only the interaction process and relationship between the database application (DBMS) and file system, but also the unique I/O behavior according to the journal mode of DBMS.

To analyze queries of real applications, we first modified DBMS's library in order to collect queries (e.g., page size, default journal mode, and query type such as insert, select, update, and delete) on the mobile device on-the-fly. Then, we gathered the real-world queries of DBMS by running popular mobile applications on a target phone (we call the collection of these queries the *SQL query*). This SQL query can help to understand the characteristics of workloads (e.g., the number of transactions and read/write ratio) and to confirm its pure performance excluding the overheads on CPU, memory, network, and storage. To fairly compare the effectiveness of each journal mode, we evaluated each journal mode by replaying the same SQL query on the DBMS of our framework. In this way, we can obtain the real performance of mobile devices without any interference. We also modified the block layer in the kernel to better understand the impact of each journal mode on mobile storage (e.g., eMMC and SDcard). We can thus observe I/O behaviors through the relevant attributes (e.g., logical block address, file name, and request size) of each I/O request. For our evaluation, we implemented the proposed evaluation framework on one of the recent smartphones, the target phone (Table II). We also developed S-WAL on DBMS that is embedded in target phone, with the compression library, which is one of the most popular compression/decompression algorithms.

B. Mobile Workloads

For comprehensive evaluation, we carefully selected four most popular mobile applications, including Mail, Messenger, Message, and SNS, and collected their SQL query for four weeks on our evaluation framework (Section IV-A). In this section, we describe the workload characteristics of each mobile application. Table III summarizes the characteristics of the SQL queries of each application.

Mail: Mail is one of the most popular applications for smartphones. This application stores all events related to email services (e.g., contents of email, senders, receivers, signatures, and label names) into one database file, mailstore.db. While collecting the SQL query, we performed a large number of relevant operations, such as sending/receiving email, changing label, and searching for keyword in the inbox. As a result,

TABLE III
CHARACTERISTICS OF MOBILE APPLICATIONS

	Mail	Messenger	Message	SNS
Default journal mode	WAL	PERSIST	PERSIST	PERSIST
# of insert queries	48946	2252	1854	25322
# of select queries	104705	2838	19547	114
# of update queries	35094	4348	6216	0
# of delete queries	23747	769	773	58
# of transactions	32392	7301	7740	15998
Avg. # of queries per transaction	3.3	1.0	1.1	1.5
Read:write ratio	5:5	3:7	7:3	0:10
Line count	296350	10611	42809	76330
Trace size	70 MB	1.7 MB	9 MB	7.2 MB

SQL query contains a large number of select and update operations. Interestingly, Mail adopts the WAL mode as its default journal mode to improve the performance of the application.

Messenger: Messenger is a popular messenger application in real-world. This application uses three independent database files to service text message and BLOB data in parallel. Unlike Mail application, Messenger uses the PERSIST mode. Also, Messenger generally creates a single transaction for processing one query (e.g., the average number of queries per transaction is 1). The ratio of read-write operations induced by this application is 3:7.

Message: The SQL query of Message is obtained from the service for short message service (SMS). A total of 6 files are generated by this application, but babel1.db is accessed most frequently to store the SMS data. Unlike Messenger application, Message has a large number of select queries (i.e., read-intensive application), while the SMS is mainly utilized for mobile marketing nowadays. However, the number of queries per transaction closely matches that of Messenger.

SNS: This SQL query is collected from SNS application that reads news feed, sends messages, and uploads photo files. SNS generally creates 20 database files to permanently store its data (e.g., newsfeed, notifications, bookmarks, contacts, and BLOB) in parallel. Therefore, this SQL query includes a large number of insert queries. Also, SNS tends to group some insert queries into a transaction even though it uses the PERSIST mode (i.e., the average number of queries per transaction is about 1.5).

C. Effectiveness of S-WAL

To understand how S-WAL positively impacts the performance of mobile devices, we carefully analyzed I/O traffic caused by DBMS. At this point, we focus on the amount of I/O traffic for each journal mode of DBMS. We compared the number of write operations incurred by each journal mode (Figure 7). As shown in Figure 7, the MEMORY mode completely eliminates the writes for updating a journal file because its journal file is kept on the DRAM memory. Meanwhile, PERSIST and WAL mode issue a large number of write operations to the underlying mobile storage for updating their database and journal file in the storage. Compared to WAL mode, PERSIST mode issues a significantly larger number of storage writes to update both its journal and database file because it leads to extra writes for initializing all used blocks to zero. Surprisingly, our results show that RBJ mode (e.g., PERSIST and MEMORY) frequently updates its database file with fsync() system call. On the other hand, WAL mode generates $2.5\times - 13.1\times$ less writes for updating the database file. Such a reduction is possible because WAL mode can reduce the number of write operations issued to the database file by performing the *page invalidation* on the WAL file.

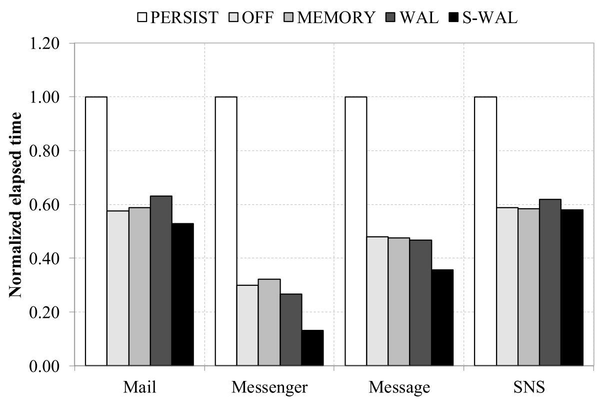

Fig. 8. The elapsed time (second) for each journal mode.

Figure 7 also shows that the database writes of the S-WAL mode are identical to those of the WAL mode. This is not surprising because S-WAL basically extends the traditional WAL mode, except for the compression and decompression on the journal data. Therefore, it also preserves its database file in mobile storage and take the benefits of page invalidation. However, S-WAL significantly reduces the number of journal writes by 4.6× on average and up to 6.4× in the best case, compared to the WAL mode. These results show that most data updates of mobile applications can be compressed and packed into a single journal write as shown in Figure 4.

D. Performance of S-WAL

In order to investigate the impact of S-WAL on the performance of real applications, we replayed the SQL queries on the target phone through our SQL-based evaluation framework. Figure 8 shows the elapsed time for each workload, comparing S-WAL mode with representative journal modes of DBMS (the results are normalized to PERSIST mode). As shown in Figure 8, S-WAL mode presents more efficient

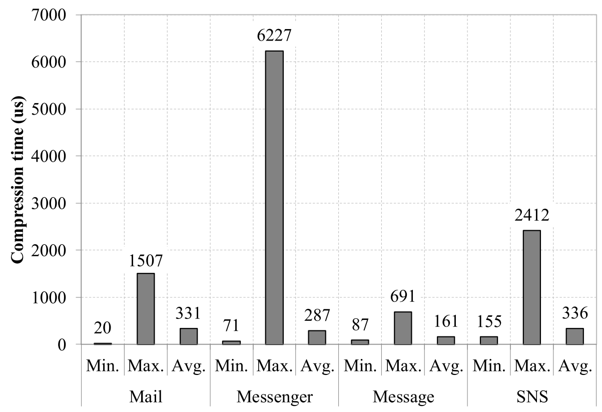

Fig. 9. Runtime overhead of S-WAL.

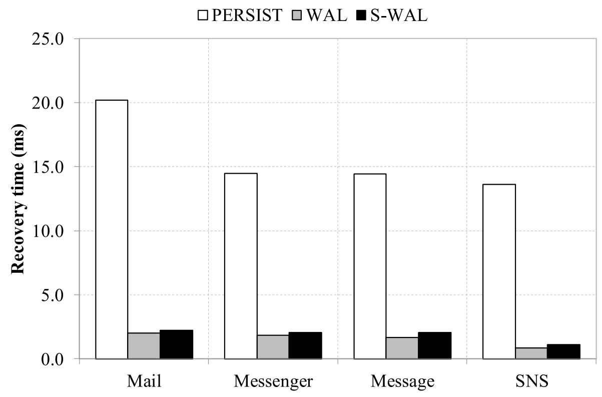

Fig. 10. Recovery overhead of S-WAL.

performance than the traditional WAL mode in all workloads. S-WAL outperforms the traditional WAL mode and PERSIST mode by 7%–51% and $1.7\times$ –$7.5\times$, respectively. This performance gain in S-WAL is primarily due to the byte-granularity compression logging that delays write operations for updating the journal file and then packs them into a single journal write via compression. On the other hand, PERSIST mode reveals the worst performance among the existing journal modes in all workloads. However, these results are not surprising because it frequently writes not only the database but also the journal file with excessive fsync() system calls as shown in Figure 7.

E. Overhead of S-WAL

In this section, we describe two extra overheads of S-WAL mode: runtime and recovery overhead.

Runtime overhead: In order to guarantee data consistency and durability at application-level, S-WAL always writes updated data twice: first to the WAL file and then to the database file. This procedure is similar to the traditional WAL mode. However, since S-WAL compresses the raw data journaled by DBMS to alleviate the redundant journaling operations and harmful write amplification, it requires time for data compression on-the-fly. We measured this compression time for inspection, while running four real-world applications. Figure 9 presents the compression latency measured on the target phone. This clearly indicates that the runtime overhead of S-WAL is negligible because the compression can be performed in less than 336 microseconds, on average.

Recovery overhead: After system failure occurs, the traditional WAL mode of DBMS only replays redo logs in the WAL file to guarantee crash consistency. On the other hand, S-WAL performs the decompression operation before replaying redo logs because it keeps redo logs in compressed form (i.e., compressed logs). We further evaluated three different journal modes for the case of system failure: PERSIST, WAL, and S-WAL mode. To verify the recovery time of each journal mode, we suddenly turned off phone while DBMS was running each application, and then we measured the time taken to recover the DBMS. For validity, we calculated the average recovery time measured from five separate runs. Figure 10 shows the average recovery time of each journal mode. As shown in Figure 10, PERSIST mode significantly suffers from long recovery time because this mode generates a large number of copy operations to roll back all consistent contents belonging to the last successful commit transaction. On the other hand, WAL and S-WAL mode only rebuild the wal-index from the consistent WAL file because the wal-index has mapping information for the checkpoint. The successfully committed compressed logs are then decompressed for updating the original database. The recovery time of S-WAL mode takes slightly longer than that of WAL mode, as shown in Figure 10, where we see that the recovery overhead of S-WAL is also negligible.

V. Related Work

In this section, we briefly describe prior work related to ours according to two categories as follows.

Performance of Mobile Devices: To improve the performance of mobile devices, most early work focused on mobile storage, which is composed of NAND-flash memories [3], because it has unique characteristics such as no in-place update, asymmetric read and write latency, and limited lifetime. Some researchers proposed a variety of flash-aware techniques in page cache or file system layer [23], [24]. Recent work has focused on the I/O behavior of mobile applications [1], [2], [4], [11], [25]. Nguyen *et al.* [25] analyzed the I/O behavior on smartphones and proposed SmartIO to reduce application I/O delay. Jeong *et al.* [1] studied the effects of asynchronous I/O on mobile devices and introduced the QASIO to minimize response latency. Jeong *et al.* [2] investigated the relationship between DBMS and the journaling of the kernel, and then reported the journaling of journal problem. They proposed a set of optimization schemes (e.g., external journaling and polling-based I/O) to resolve the problem. Meanwhile, some researchers have proposed DBMS enhancement techniques such as LS-MVBT [11] and WALDIO [4], which demonstrated significant performance gains. While these performance gains are larger than those of S-WAL, such techniques were evaluated only on synthetic workloads. Therefore, the performance gains of S-WAL are fundamentally different from those of previous techniques, and have significant meaning in terms of practicability because we evaluated S-WAL on the real-world workloads (Table III), which represent popular mobile applications.

Database Compression: The compression of data is widely used in database engines because it helps improve the performance of the database by reducing the storage space and I/O bandwidth. Therefore, in most database engines [26], a lossless compression mechanism is adopted to compress the database. Compression mechanisms can be classified into two types: row-oriented compression and column-oriented compression. DBMS engines based on row-oriented storage model can compress their database at a page granularity or table granularity. On the other hand, some database applications support compression of its database at a column granularity with the compression algorithm. The compression algorithm of S-WAL is considerably different from existing database compression algorithms as we compress only the journal content instead of the database.

VI. Conclusion

In this paper, we analyze the existing journal modes of database application and compare the performance of each journal mode to find the most efficient journal mode for mobile devices. Based on our observation, we propose a novel journal mode, the S-WAL mode, that extends the traditional WAL mode to exploit the *flash-friendly writes* and *page invalidation advantages*, while guaranteeing data consistency and durability at application-level. We believe that S-WAL is a practical way to support application-level crash consistency on the existing mobile devices for two reasons. First, S-WAL is designed to avoid any modifications of not only Linux kernel but also source code of applications, and it requires only about 40 lines of code changes on the code of traditional WAL mode of DBMS. Second, unlike some of the existing proposed approaches, S-WAL never requires any special hardware support.

References

[1] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," in *Proc. USENIX FAST*, 2015, pp. 191–202.

[2] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX ATC*, 2013, pp. 309–320.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. USENIX FAST*, 2012, pp. 1–14.

[4] W. Lee *et al.*, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proc. USENIX ATC*, 2015, pp. 235–247.

[5] *SQLite*. Accessed: Nov. 21, 2015. [Online]. Available: http://www.sqlite.org

[6] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," in *Proc. ACM VLDB*, 2015, pp. 1454–1465.

[7] D. Campello, H. Lopez, L. Useche, R. Koller, and R. Rangaswami, "Non-blocking writes to files," in *Proc. USENIX FAST*, 2015, pp. 151–165.

[8] D. Q. Tuan, S. Cheon, and Y. Won, "On the IO characteristics of the SQLite transactions," in *Proc. IEEE MOBILESoft*, Austin, TX, USA, 2016, pp. 214–224.

[9] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, "Reducing solid-state storage device write stress through opportunistic in-place delta compression," in *Proc. USENIX FAST*, 2016, pp. 111–122.

[10] J. Kim, C. Min, and Y. I. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 217–224, May 2014.

[11] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in Android I/O: Multi-version B-tree with lazy split," in *Proc. USENIX FAST*, 2014, pp. 273–285.

[12] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight application-level crash consistency on transactional flash storage," in *Proc. USENIX ATC*, 2015, pp. 221–234.

[13] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *Proc. USENIX FAST*, 2014, pp. 287–293.

[14] S. Park, T. Kelly, and K. Shen, "Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data," in *Proc. ACM EuroSys*, 2013, pp. 225–238.

[15] R. Verma *et al.*, "Failure-atomic updates of application data in a Linux file system," in *Proc. USENIX FAST*, 2015, pp. 203–211.

[16] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *Proc. ACM ASPLOS*, 2016, pp. 385–398.

[17] 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*, 2013, pp. 97–108.

[18] *SQLite: Write-Ahead Logging*. Accessed: Oct. 19, 2015. [Online]. Available: http://www.sqlite.org/wal.html

[19] *AndroBench*. Accessed: Jan. 11, 2016. [Online]. Available: http://www.androbench.org

[20] *MobiBench*. Accessed: Oct. 10, 2015. [Online]. Available: http://www.mobibench.co.kr

[21] D. H. Kang and Y. I. Eom, "FSLRU: A page cache algorithm for mobile devices with hybrid memory architecture," *IEEE Trans. Consum. Electron.*, vol. 62, no. 2, pp. 136–143, May 2016.

[22] *SQLite: Database File Format*. Accessed: Jan. 3, 2016. [Online]. Available: http://www.sqlite.org/fileformat.html

[23] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. ACM SIGMETRICS*, 2012, pp. 235–246.

[24] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. USENIX FAST*, 2015, pp. 273–286.

[25] D. T. Nguyen *et al.*, "Reducing smartphone application delay through read/write isolation," in *Proc. ACM MobiSys*, 2015, pp. 287–300.

[26] *MySQL 5.7 Reference Manual*. Accessed: Nov. 28, 2015. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en

Dong Hyun Kang received the B.S. degree from the Department of Computer Engineering, Korea Polytechnic University, South Korea, in 2007 and the M.S. degree from the College of Information and Communication Engineering, Sungkyunkwan University, South Korea, in 2010, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. His research interests include file and storage systems, operating systems, and embedded systems.

Woonhak Kang received the B.S. degree in computer engineering and the M.S. and Ph.D. degrees in electrical and computer engineering from Sungkyunkwan University, South Korea, in 2006, 2010, and 2016, respectively. He was a Post-Doctoral Research Fellow with the College of Computing, Georgia Institute of Technology from 2016 to 2018. He is a Research Scientist with eBay, San Jose, CA, USA. His research interests include database management systems, flash-based storage systems, distributed systems, and key-value store.

Young Ik Eom received the B.S., M.S., and Ph.D. degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. He was a Visiting Scholar with the Department of Information and Computer Science, University of California at Irvine from 2000 to 2001. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. His research interests include virtualization, operating systems, storage systems, cloud systems, and UI/UX system.