SJM: an SCM-based Journaling Mechanism with Write Reduction for File Systems

Lingfang Zeng[†], Binbing Hou[‡], Dan Feng[†], Kenneth B. Kent[*]
[†] Wuhan National Laboratory for Optoelectronics,China 430074
[†] School of Computer, Huazhong University of Science and Technology,China 430074
[‡] Department of Computer Science and Engineering, Louisiana State University, USA, Baton Rouge, LA 70803
[*] Faculty of Computer Science, University of New Brunswick, Canada E3B 5A3
{lfzeng,dfeng}@hust.edu.cn, bhou@csc.lsu.edu, ken@unb.ca

ABSTRACT

Considering the unique characteristics of storage class memory (SCM), such as non-volatility, fast access speed, byte-addressability, low-energy consumption, and in-place modification support, we investigated the features of over-write and append-write and propose a safe and write-efficient SCM-based journaling mechanism for a file system called SJM. SJM integrates the ordered and journaling modes of the traditional journaling mechanisms by storing the metadata and over-write data in the SCM-based logging device as a write-ahead log and strictly controlling the data flow. SJM writes back the valid log blocks to the file system according to their access frequency and sequentiality and thus improves the write performance. We implemented SJM on Linux 3.12 with ext2, which has no journal mechanisms. Evaluation results show that ext2 with SJM outperforms ext3 with a ramdisk-based journaling device while keeping the version consistency, especially under workloads with large write requests.

Keywords

Storage Class Memory, File System, Data Consistency, Journaling

1. INTRODUCTION

Maintaining the consistency of file systems is an important research topic within the field of storage systems. In order to quickly restore file systems to a consistent state after unexpected system crashes, journaling mechanisms are widely adopted for their simple implementation and good performance. However, with the rapid development of emerging materials and storage technologies, storage class memory (SCM) [7, 6] plays an increasingly important role in storage systems for its excellent DRAM-like access performance and disk-like non-volatile characteristics. In recent years, more and more new storage devices, such as PCM (Phase Change Memory) [14], STT-MRAM [11] and Memristor [22], support byte level access and local (in-place) update, have higher reading and writing performance, and higher energy efficiency. These devices will most likely become next generation SCM technology solutions. However, one cannot take full advantage of SCM by simply utilizing it as a journaling device because traditional journaling mechanisms are designed for disk-based devices.

Logging technology is a commonly used method to ensure the consistency of file systems, however, it introduces another problem. Journaling mechanisms face a "write twice" problem: before changing the file system, the first step is to update the record in the log. Therefore, file systems usually only focus on logging file system metadata, such as ext3fs [23] and Reiserfs [20]. The log is normally stored in the disk partition or log file. When the system is down, we can use the log records, which are written back to disk sequentially, to recovery the system. The traditional logging technology in the system restores indiscriminately so that multiple log copies of the same piece of data will occupy more than one log space. System recovery should also write back many times, this not only occupies a large amount of log space, but also extends the system recovery time. Shen et al. [21] studied the cost of additional file system journaling and found the significant cost (slowdown) is up to 73% due to implementation limitations of the current system.

Aiming at these problems, this paper proposes a safe and write-efficient logging method, SJM (SCM-based Journaling Mechanism), to make full use of the performance advantages and byte addressing and modifying characteristics of SCM. The SCM device is mounted on the memory bus as the log device. Using the logging method does not require different types of data requests. In view of the phenomenon that multiple log blocks of a certain file system block usually co-exist in the logging device, SJM only maintains the log blocks of the latest version; Write back of the valid log blocks to the file system is according to their access frequency and sequentiality. In addition, because the journaling mechanism of SJM adopts the physical log mode, each data block update will generate a log block, so it will create a relatively large data log for small data write requests. In view of this situation, the SJM uses a delay recycling approach, under the premise of ensuring the integrity of the transaction log by delaying the log block copying to allow modification of the old log block, thus, reducing as much as possible writing to the log data flow.

DISCS-2015, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2831244.2831246.

The rest of the paper is organized as follows: Section 2 discusses the related work; Section 3 presents the design of SJM and the detailed implementation; Section 4 provides an evaluation of the mechanism in a real system implementation; Section 5 concludes the paper.

2. RELATED WORK

2.1 File system consistency

Timely restoration of the file system to a consistent state and the resumption of normal service following unexpected downtime continues to be a core issue and important research topic in the study of file systems. General operating systems provide a file system check tool (e.g. *fsck*), capable of scanning the whole file system metadata in downtime, correcting possible inconsistencies, and returning it back to a consistent state. However, this repair process takes a long time to complete and often requires several hours or days. For storage systems requiring rapid response time and continuous provision of services, long periods of service interruption can cause huge losses of property and maybe even customers, which is obviously intolerable.

People have put forward many positive methods to ensure the file system consistency by considering the possible inconsistencies after downtime when the system is in operation. The file system can quickly recover to a consistent state after unexpected downtime by guaranteeing the atomicity of file system operations through certain mechanisms. For example, positive consistency methods mainly include journaling mechanisms [24], CoW (Copy-on-Write) [10] and soft updates [19]. Among them, journaling mechanisms are widely adopted by many file systems for their simple implementation and good performance.

2.2 Based on SCM

With the emergence of the next generation of storage with excellent performance characteristics, such as SCM, much research is being conducted into applications of SCM in all kinds of fields. Important among the work are more efficiently guaranteeing the consistency of the storage system after the introduction of SCM and accelerating the storage system using SCM. Research in these areas are of two main types.

2.2.1 *The consistency of a storage system based on SCM*

Condit et al. proposed the CoW (Copy-on-Write) file system called BPFS for byte-addressable storage [5]. BPFS performs an in-place-write when the updated data size is smaller than the unit of an atomic operation. This can reduce the out-place-update overhead of CoW significantly. Wu et al. suggested a file system for SCM [27]. Assuming that SCM resides on the memory bus and can be accessed directly from the CPU, they proposed the file system access files through the same address space of virtual memory systems. Based on PCM, Lee et al. [17] presented the *Shortcut-JFS*, which reduces the amount of journaling by more than half by exploiting the byte-accessibility of PCM. Shortcut-JFS performs differential logging and in-place checkpointing to remove the unnecessary overhead of block copying.

Lu et al. proposed LOC (Loose-Ordering Consistency) [18] to satisfy the order requirements of persistent memory writes. By eager commit, LOC reduces the commit overhead for writes within a transaction. LOC relaxes the ordering of writes between transactions by speculative persistence. Chidambaram et al. [4] presented NoFS, which provides crash consistency without ordering writes using a back pointer-based consistency mechanism.

Venkataraman et al. [25] presented CDDSs (Consistent and Durable Data Structures) to allow programmers to safely exploit the low-latency and non-volatile aspects of new memory technologies. CDDSs use versioning to allow atomic updates without requiring logging. The same versioning scheme also enables rollback for failure recovery.

By combining a logging device with cache, UBJ [15] uses a double circular linked list of all kinds of transactions in a JBD (Journaling Block Device) memory transaction management buffer, to ensure the consistency in logic. We will demonstrate that, compared to the UBJ, SJM is better in inspecting access characteristics of files and using the characteristics of the SCM to do XOR updates to reduce write operations.

2.2.2 *Accelerated storage system using SCM*

In this case, by combining the SCM device with the storage system and using the non-volatile characteristics of SCM, one can simplify and accelerate the storage system. For example, Lee et al. [16] presented a versioning file system for PCM that reduces the writing overhead of a snapshot significantly by breaking the recursive update chain at the immediate parent level. Fang et al. [6] presented an SCM-based approach for DBMSs logging, which achieves higher performance by using a simplified system design and better concurrency support. This logging approach is used to replace the traditional disk based logging approach in DBMSs.

Chen et al. [3] proposed FSMAC to optimize metadata access by exploiting the advantages of Nonvolatile Memory (NVM). FSMAC decouples the data and metadata I/O path, by placing the data on disk and the metadata in NVM at runtime. Thus, data is accessed in blocks from the I/O bus and metadata is accessed in a byte-addressable manner from the memory bus. Metadata access is significantly accelerated and metadata I/O is eliminated because metadata in the NVM is on longer flushed back to disk periodically. A light-weight consistency mechanism combining fine-grained versioning and transactions is introduced in the FSMAC.

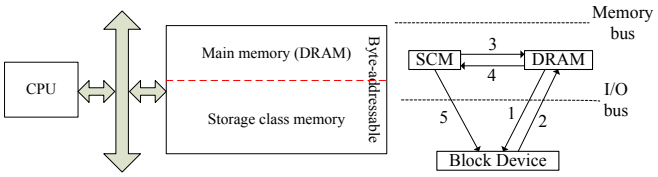
Lee et al. [15] presented a buffer cache architecture that subsumes the functionality of caching and journaling by making use of non-volatile memory. The proposed in-place commit scheme avoids logging, but still provides the same journaling effect by simply altering the state of the cached block to frozen. Chen et al. [2] adopted a hybrid storage model, called PMBD, which directly accesses persistent memory (e.g. SCM) attached to the memory bus and exposes a logical block I/O interface to users.

The above mentioned studies proposed several SCM-based approaches to reduce the journaling overhead. However, traditional journaling mechanisms are optimized for block-level accesses while the byte-accessibility characteristic of SCM is not explored and exploited. Moreover, it may not be easily applicable to a hybrid storage system consisting of hard disk drives (or solid state drives) and SCM, which, we believe, will be the major form of leveraging SCM in storage systems in the near future.

3. SJM DESIGN AND IMPLEMENTATION

3.1 SJM system overview

Given that many kinds of SCM have wear limits that shorten their lifespan, we want to reduce the write transactions to SCM. Therefore, we need an interface to support byte accesses between memory and storage devices. Although SCM is not commercially available on a large scale, most research assumes that SCM will be placed in standard DIMM slots. This is a reasonable assumption as we need to utilize the byte-accessibility of the SCM. However, this architecture has a weakness in that it limits the storage capacity according to the number of DIMM slots. PCI express is another feasible option to connect SCM to a computer system. Because the PCI express bus shields the characteristics of SCM byte addressing, any changes to a data block in SCM will involve modifying the entire block of data. One aspect of this write amplification is an increase in write traffic, but it also increases the wear time of the SCM. So, in the design of SJM, we will mount the SCM on the memory bus. On one hand, the access speed of SCM will be close to DRAM and we can utilize its rapid access characteristics more effectively, instead of spending too much time on the bus delay; on the other hand, when the SCM acts as a logging device, there is a very close interaction relationship with memory, so mounting the SCM on the memory bus can make full use of its characteristics of byte addressing, thus we can develop more efficient journaling mechanisms. In addition, when the SCM is used as a logging device only, we require a very small storage capacity, so the amount of DIMM slots consumed on the main board does not become a serious limitation. This paper assumes the operating system itself already provides the write protection mechanism and the prohibition of memory rearrangement mechanism, rather than repeat this work in this research.



(a) System module (b) Data flow

Figure 1: SJM overview.

Fig. 1(a) shows a possible module using SCM in our study. As is shown in Fig. 1(b), the SCM and DRAM are mounted on the memory bus in parallel. Data flow 1 and data flow 2 depict DRAM interaction with the file system in the absence of a journaling mechanism. Data flow 3 and 4 shows the interaction between DRAM and SCM in SJM. Data flow 4 shows that the process of memory transaction writes to the logging device, and data flow 3 shows the data recovery process from the logging device after downtime. Data flow 5 indicates that, at the appropriate time, the valid log records will be effectively written to the file system.

3.2 SJM log mode

The Journaling Block Device (JBD) [13] provides a file system-independent interface for file system journaling. JBD log mode is divided into three kinds: writeback mode (*data = writeback*), ordered mode (*data = ordered*) and journaling mode (*data = journal*). Among them, the *writeback* mode log only contains metadata, and is only able to ensure the metadata consistency of the file system; a journaling mode log record contains all of the data and metadata, so it can ensure the version consistency of the file system. The ordered pattern is a compromise: it only records the metadata schema, but strictly ensures the data is written into the file system before writing the metadata log. Thus, for an append-write, an ordered mode can ensure version consistency of the file system; while for over-write, an ordered pattern can only guarantee the data consistency of the file system.

The write operation can generally be classified into three types: (**i**) append-write, (**ii**) over-write and (**iii**) a part is append-written, another part is an over-written. Considering the practical application, the frequency of an append-write is much greater than an over-write. Therefore, SJM will write-ahead of the over-write data or the part of over-write data in the write operation based on an ordered pattern of JBD, then reduce the log submit as much as possible to ensure the version consistency of the file system.

Compared with JBD's journaling mode, which only records the over-write data and metadata, SJM can greatly reduce the log submissions, and also reduce the "write twice" overhead of the journaling mechanism, while saving log space, and reducing the frequency of log data written back to the file system. All update operations in the file system involve metadata modifications, including file modification, creation, deletion, copying, moving and so on. Updating the file system metadata involves a large number of random small write operations, for example, one update operation involving the bitmap may only involve 1 bit or several bits, a file inode accounts for only 64 bytes, so to update an inode also involves modification to only a few bytes. Therefore, by reducing the write-back frequency of metadata, we can, thereby improve the write performance of the file system.

3.3 SJM transaction mechanism

SJM can distinguish between different types of write operations, so a transaction in SJM contains two bidirectional circular linked lists: a two-way cycle chain manages the buffer of append-write data; another two-way cycle chain manages the metadata buffer and the over-write data buffer. Considering that the access speed of SCM is close to that of memory, when a log has completed writing to a SCM device, transactions can be written back to the file system directly from the logging device, without having to maintain a checkpoint chain like the JBD logging mechanism. (Note: The JBD mechanism deletes the corresponding log records from the logging device after the checkpoint chain data is written to the file system.)

The SJM transaction mechanism includes two kinds of transactions: running transactions, and committing transactions. Because the append-write transaction buffer in the chain requires synchronization to the file system, a committing transaction often contains multiple completed running transactions to improve writing efficiency.

The normal operation in Fig. 2 shows the interaction between DRAM, the logging device and the file system, which mainly includes three processes. **Process 1** (P1) is an append-write data synchronization process, which synchronizes the buffer data of the append-write link in committing

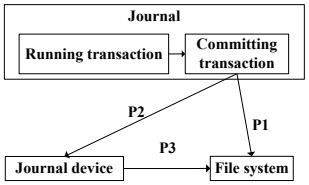


Figure 2: SJM transaction mechanism.

transactions to the file system. **Process 2** (P2) is a write log process in which the metadata and the buffer of over-write data in committing transactions are written to the logging device. **Process 3** (P3) is a log write-back process, which, at the appropriate time, synchronizes the valid log record in the logging device to the file system. Considering the importance of the update sequence for file system consistency, the data flows of *P1*, *P2* and *P3* must ensure that the order is $P1- > P2- > P3$. Note in particular, the sequence of *P1* and *P2* refers to a logical sequence and not the actual data flow sequence. The process to write the log may begin before a process of append-write data synchronization, but it can be marked as the write log ends after all append-write data in the transactions are synchronized to the file system. In other words, even if the transactions of all metadata and append-write data have been submitted to the journal, the system cannot submit the next transaction to the log device until after the process of append-write data synchronization is completed.

3.4 SJM space management

SJM makes full use of the proximity of SCM to the DRAM, as well as the checkpoint transaction and the related operations that are sustained in the JBD mechanism to transfer to the logging device. Hence, SCM becomes the key to manage the committing transaction and the checkpointing transaction for the SJM space management.

SJM is a journaling mechanism based on SCM, so the efficiency of sequential write and random write are not significantly different. SJM only needs to maintain the logical order and does not require guaranteeing its physical consistency.

3.4.1 SJM logical layout

As shown in Fig. 3, the SJM can be logically divided into three parts: *checkpoint area*, *commit area*, and *free area*. The areas are identified by using the pointer of the *superblock*.

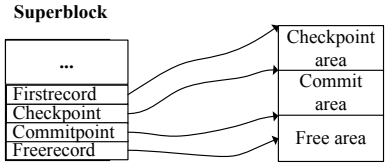


Figure 3: SJM logical layout.

In the *free area*, a data block is a free block and can be used to record the log data. The *freerecord* pointer references the first free log record in all the log space.

All log records in the *checkpoint area* are in a consistent state and all devices have only stored the most recent copy. The *checkpoint* pointer references the last block of the checkpoint area.

When a logging transaction begins, the log record writes along the checkpoint area of the last completed transaction. When all the log blocks of this transaction are finished, the *commitpoint* pointer references the last log record of the *commit area*. Then each log record is traversed in the *commit area*. If the blocks in the *checkpoint area* have an old copy, then it is marked as invalid. When all log records in this transaction are completed, the *commitpoint* pointer references the position of *checkpoint*.

3.4.2 SJM physical layout

Since SJM is based on SCM which is mounted on the memory bus, SJM can make full use of the byte addressing characteristics of SCM. The system directly uses the pointer to mark the start and end of a write transaction. There is no need to submit blocks to represent the transaction commit completed as required in the JBD mechanism.

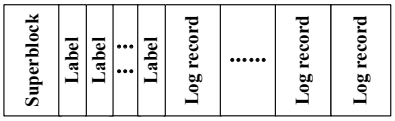


Figure 4: SJM physical layout.

In Fig. 4, the SJM log space can be divided into three successive parts: super block (*Superblock*), log record label (*Label*) and log record (*Logrecord*). The super block is used to record the related information of the SCM logging device including the magic number, the first available log block, the total number of log records, the number of invalid log records, the effective number of log records, the available log space, the transaction record mark, which marks the current transaction submission complete, and the next available log block, which is used for log space allocation.

Different from the JBD mechanism, SJM strictly controls the logical sequence of log writes. A logical sequence of SJM is ensured by means of log tags. Every time the system allocates space for each write log, it realizes the logical order by the allocation of successive physical tags. It realizes the logical recovery by recycling tags when dealing with garbage collection for space. As such, the corresponding physical block indicated by these tags may be random in physical space.

3.5 XOR update scheme

As described in subsections 3.2 and 3.3, in order to ensure version consistency of file systems with minimal data copying and a reduction of the log writes on the system level, SJM integrates *ordered mode* and *journaling mode*. In addition, taking the characteristics of byte-addressing and support for local modification of SCM into account, SJM uses the old log version further to reduce log writes.

SCM contains both a metadata and an over-write data log. According to the locality principle, this data will be updated again in a short time. Especially the metadata, which will be accessed frequently [3]. In the log space, there is more than one log version of data blocks. Considering the difference between the log blocks, it may be tens of bytes or a few bytes (e.g. update bitmap block, update inode), so there is no doubt that reducing log writes by directly modifying

the old log version is advantageous. To change an old log version, we simply need to write the differences between the two data blocks - generally a few bytes or tens of bytes. To write the log block directly would require writing the whole data block that is usually 4KB.

Assume that a data block number is D and the size of the data block is 4KB in a file system. The data block D is first modified and a log block Ll is generated, $L1 = D + delta\_1$, where $delta\_1$ is the first amendment part. The data block D is subsequently modified and a log block $L2$ is generated, $L2 = L1 + delta\_2$, where $delta\_2$ is the second amendment part. At this time, $L2$ is the latest version for data block D. $L1$ is the most recent stale log version and should be marked as invalid. The data block D is further modified and a log block $L3$ is generated, $L3 = L2 + delta\_3$, where $delta\_3$ is the third amendment part. The system would generate a 4KB log write if it were to directly write the log block. The system only needs to write the update part if it is based on the old log version.

As the most recent log copy, $L2$ cannot be directly modified. If the system modifies $L2$ directly, the updating interrupt will damage the integrity and consistency of the log transactions. However, for $L1$ which is the most recent stale copy and has been marked as invalid, even if the update process is interrupted, the consistency and integrity of the log transaction will not be damaged.

From the above analysis, $L3 = L2 + delta\_3 = (L1 + delta\_2) + delta\_3 = L1 + (delta\_2 + delta\_3)$. Therefore, the system only needs to update the $delta\_2 + delta\_3$ based on $L1$. Now the question is how to determine the corresponding update quantity for each log block. For this, SJM adopts a XOR update scheme.

The most direct way to determine the difference between two data blocks is a XOR operation. The basic principle of XOR is that equal data is 0, dissimilarity is 1. For the log block $L1$ which is in the log device and the log block $L3$ which is about to write to the log device, the system only needs to read the log block $L1$ and $L3$, then perform a XOR operation to obtain the relationship block P ($P = L1 \oplus L3$).

$L1(i)$ represents the i-th bytes of log block $L1$, $L3(i)$ represents the i-th bytes of log block $L3$, and $P(i)$ represents the i-th bytes of relationship block P.

We analyze the relationship between the blocks $L1$ and $L3$: if $P(i)$ is all zero bytes, and the i-th byte of the log block $L3$ and log block $L1$ are the same, and the system does not update; if $P(i)$ is not all zero bytes, then the i-th byte of the log block $L3$ and log block $L1$ are different, and the XOR operation between $P(i)$ and $L1(i)$ will update the log block $L1(i)$: $L1(i) = P(i) \oplus L1(i)$; or directly assign $L3(i)$ to the $L1(i)$: $L1(i) = L3(i)$.

4. PERFORMANCE EVALUATIONS

4.1 Experimental setup and methodology

The performance evaluation was conducted on a PC platform with an Intel Pentium(R) dual-core 2.93GHz processor and 1GB DDR memory. In the system, the HDD is a WD5000AADS-00S9B0 500GB SATA disk. The experimental setup is shown in Table 1. The test software is IOZone [1], PostMark [12] and Filebench [26] respectively. We mainly show the write performance of the three benchmarks because we focused on reducing the journaling overhead by using the SJM mechanism.

Table 1: Experimental setup.

Machine	CPU: Pentium(R), Dual-Core 2.93GHz
OS	Fedora 17, Linux 3.12
RAM	1GB DDR
Disk driver	WD5000AADS-00S9B0 500GB HDD
Benchmark	IOZone(version 3.4.0) [1] PostMark (version 1.51) [12] Filebench(version 1.4.9) [26]

Since an SCM device is not yet commercially available and read and write performance of an SCM device is close to memory, most research in the domain uses memory to simulate SCM [17, 16, 25, 6, 8, 9]. Our tests of the SJM also used memory to simulate the SCM. There are many methods available to acquire the memory to simulate the SCM, such as dividing an independent memory zone in memory, using a shared memory method, or memory mapping (MMAP). Before memory initialization, SJM uses *alloc_bootmem* to reserve a 128MB block of contiguous physical memory that is referenced by the pointer *scm_addr*. This partition of physical memory is fully used and managed by the user, which bypasses the Linux memory management mechanism.

For the sake of fairness, we compare our system (shown as $ext2\_SJM$) with ext3fs that uses the same capacity ramdisk as its journaling device and adopts the JBD mechanism (shown as $ext3\_JBD$), and with the original ext2fs without journaling (shown as $ext2\_no$). The log mode of the control group $ext3\_JBD$ is set to be *data = journal*. In addition, although the Linux operating system provides the ramdisk tool, in order to further reduce the performance impact the equipment itself brings, ramdisk is formerly used to simulate the SCM which has an added layer of block device driver.

4.2 Large file test

IOZone [1] is a set of open source testing tools for file systems. IOZone completes the performance test of a file system by performing a series of I/O operations. When testing IOZone, the file size should exceed the memory size. It is recommended that using a file size that is twice as large as the memory size is best. We determined the available memory size to be 231MB by using the instruction *free -m* before testing the system. This test used IOZone to measure the performance of large file writes. The size of test files used were 512MB, 1GB and 2GB respectively, while the record size was 64KB.

The write operation of IOZone creates a file and sequentially writes data to the file until the specified file size is reached. Fig. 5 shows the write performance comparisons with different journaling mechanisms. As the file size increases, write performance of $ext3\_JBD$ is relatively stable, while the write performance of $ext2\_SJM$ and $ext2\_no$ all decreased. This is because the JBD mechanism for the journal mode, no matter how large the file grows, submits transactions to the ramdisk log first, and then writes the checkpoint transaction to the file system. In this manner, the write operation of a file will be divided into many such log submissions and write checkpoint transaction operations. For $ext2\_no$, each write operation writes data to the file system, and then the metadata writes back to the file system at the right time. Because IOZone file writing is divided into several records based on the unit size of a record, then the bigger the file, the more metadata updates generated,

leading to the write back of $ext2\_no$ metadata more frequently. So the write performance of $ext2\_no$ falls away. But for $ext2\_SJM$, all write operations are append-write, so the write operation consists of only writing to the metadata log. The larger the file, the more metadata log records generated, the larger overhead of the metadata log, so the write performance of $ext2\_SJM$ decreases with a file size increase.

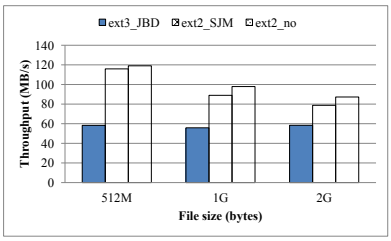


Figure 5: Write performance comparison of different mechanisms for large file size.

It can be seen from Fig. 5, for write operations of a large file, the performance of $ext2\_no$ is the highest, $ext2\_SJM$ is the second, and $ext3\_JBD$ is the lowest. Because $ext2\_no$ does not have additional journaling overhead, write performance of $ext2\_no$ is best; and $ext3\_JBD$ is the other extreme, where each write operation will maintain a write-ahead log, so $ext3\_JBD$ has the lowest performance. Although $ext2\_SJM$ also incurs a log overhead, it is only the metadata log. In combination with the garbage collection mechanism and the XOR update method we further reduce the metadata log overhead such that the write speed of $ext2\_SJM$ is slightly slower than $ext2\_no$, but is much faster than $ext3\_JBD$. The rewrite operation of IOZone performs over-write on an opened file. Compared with the write operation, it avoids the process of creating the file and data block allocation.

A performance comparison of the write and rewrite operations in the $ext2\_SJM$ mechanism is shown in Fig. 6. For $ext2\_SJM$, a write operation only needs to do the metadata log while a rewrite operation needs to do the entire data (data and metadata) log, which are the two extremes of the SJM mechanism. Seen from the graph, it is evident that the speed of a write operation is much faster than that of a rewrite in $ext2\_SJM$. For $ext2\_SJM$, although we avoid the cost of file creation and data block allocation, all rewrite operations are over-writes, so the $SJM\_rewrite$ operation incurs roughly a $2x$ cost, as such it leads to a sharp decline in writing performance.

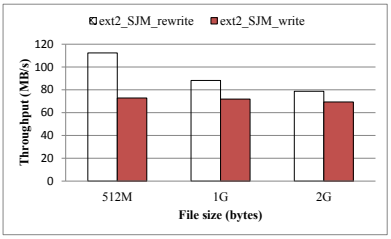


Figure 6: Write/rewrite performance comparison of $ext2\_SJM$ mechanism for large file size.

Fig. 7 shows the comparison of rewrite performance in different journaling mechanisms. Comparing Fig. 5 with Fig. 7 it can be found that the rewrite operation speed of $ext3\_JBD$ and $ext2\_no$ is faster than the write operation. This is because the rewrite in comparison to the write operation eliminates the file creation and data allocation process. In addition, the rewrite operation speed of SJM is still higher than that of $ext3\_JBD$, but the higher amplitude has been greatly reduced. Also, as the file size increases, the higher amplitude gradually decreases. When the file size is 2GB, the rewrite speed of $ext2\_SJM$ is almost equal to that of $ext3\_JBD$. This is partly because $ext2\_SJM$ in the face of rewrite operations, like $ext3\_JBD$, also faces "write two times". Therefore, the rewrite operating speed of $ext2\_SJM$ is far lower than the write operating speed, while the $ext3\_JBD$ rewrite operation was higher than that of the write operation speed, so the relative speed advantage of a $ext2\_SJM$ rewrite is reduced. On the other hand, SJM employs the write-back strategy to optimize the write-back of log data, so the speed of $ext2\_SJM$ is still higher than that of $ext3\_JBD$. At the same time, with the increased size of rewrite files, rewrite operations involving the metadata update is increased, and the rewrite log overhead is greater. So as the file size increases, the rewrite operating speed advantage of $ext2\_SJM$ decreases, so that when the file size is 2GB, the rewrite operation speed of $ext2\_SJM$ is almost equal to that of the $ext3\_JBD$.

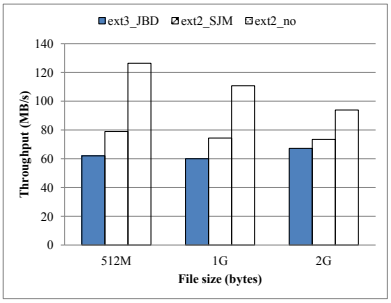


Figure 7: Rewrite performance comparison of different mechanisms for large file sizes.

4.3 Small file test

PostMark [12] is software for testing back-end storage performance. Postmark is mainly used for testing the performance of a file system under applications that require a significant number of random accesses to small files. This class of applications include e-commerce systems, mail systems, etc. The basic principle of Postmark is to create a test file pool which can set a number of files and range of sizes; do a series of transaction operations on the file pool; and finally output the test structure. Unlike IOZone, Postmark can compensate for the influence of the file system cache by adjusting the proportion of create (or delete) operations and read (or append) operations, so Postmark is suitable for the performance testing of small files. This paper uses PostMark with the settings: transactions=$1,000$ and number=$100,000$. This means the transaction number is $1,000$ and the number of files is $100,000$. Test file sizes are 4KB, 8KB, 16KB, and 32KB respectively.

As the test results in Fig. 8 show, with the increase in size of the file, write performance of $ext3\_JBD$, $ext2\_SJM$

and *ext2_no* are increased. This is because as the file size grows, the proportion of data updates in the write process is greater. The updating of data is mostly sequential updates, and metadata updates are small random writes. Therefore, the greater the file size, the greater the proportion of writes, and the write performance of *ext3_JBD*, *ext2_SJM*, and *ext2_no* are also higher.

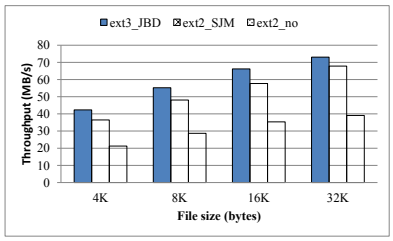


Figure 8: Write performance comparison of different mechanisms for small file sizes.

As can be seen from the graph, in a small file write test, write performance of *ext2_no* is lowest, *ext3_JBD* is the highest, *ext2_SJM* is in the middle. This is because the metadata write operation is intensive when writing small files and metadata updates occupy a very large proportion of the write operations. The *ext2_no* first writes the data, and then writes the metadata, so this mechanism breaks up the metadata update and the data update. So each write back of data results in small random write operations, write back metadata is also random small write operations, so write performance of small files for *ext2_no* provides the poorest performance. Journal mode of *ext3_JBD*, though it will bring twice the updates, will merge many write operations into one large transaction to commit back. The checkpoint transaction to the file system is also a large write, so write performance of a small file for *ext3_JBD* provides the best performance. For *ext2_SJM*, the journaling mechanism reduces the write back frequency of metadata, but the sequence control of SJM, namely performing an over-write for data and then writing the metadata log, reduces the performance of the file system.

4.4 Different load test

Filebench [26] is an automatic testing tool for the performance of file systems by performing a fast simulation of a real system load. Unlike traditional IOZone and PostMark tools Filebench can simulate various typical loads, such as a *fileserver* and *varmail*, while allowing the user to modify parameters of the load generated. The *varmail* load of Filebench is used for simulating access of a user to the server in a simulated mail system. As such, the access requests in the *varmail* load consists mostly of small amounts of random data. The *fileserver* in Filebench is used to simulate access from a user to the server in a file server system. The access requests of users in *fileserver* is mainly for large amounts of data. Therefore, we use *varmail* to simulate small data access loads and *fileserver* to simulate large data access loads. In this test, parameter settings are as follows: average dir width: 1,000; the average directory depth: 0.5-0.8, file number: 10,000; and the number of user threads: 16. The average file size is 2MB.

As Fig. 9 shows, under the *fileserver* load, write speed of *ext2_SJM* is faster than *ext3_JBD* by roughly 45%, but slightly lower than *ext2_no*. This is because most of the requests are large data requests in the *fileserver* load. The performance of *ext3_JBD* is decreased as a result of the log overhead. Comparing *ext2_SJM* with *ext2_no*, the frequency of metadata write-backs is reduced, but there is also an increase in the overhead of the metadata log, so the performance of *ext2_SJM* is slightly lower than *ext2_no*.

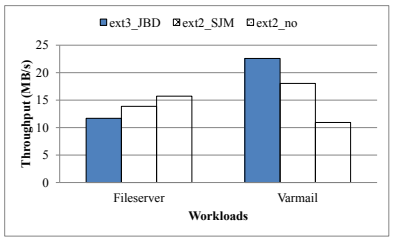


Figure 9: Write performance comparison of different mechanisms for different workloads.

Under the load of *varmail*, the write performance of *ext2_no* is the highest, *ext3_JBD* is the lowest, and *ext2_SJM* is in the middle. This is because, the write requests are mostly small in the *varmail* load, therefore the *varmail* load is metadata intensive. Because *ext3_JBD* posesses the function of merging small writes into larger writes, the write performance of *ext3_JBD* is the best. *ext2_no* faces frequent refreshing of metadata, so the write performance of *ext2_no* is the lowest. While *ext2_SJM* reduces the frequency of metadata write backs due to the effects of cache logs, but at the same time sequence control reduces performance of data writes, so write performance of *ext2_SJM* is higher than *ext2_no* and lower than *ext3_JBD*.

5. CONCLUSIONS

As the next generation of memory, SCM has the characteristics of both memory and disk. Considering the application of SCM in the storage system, the most basic idea is to replace traditional memory (including memory, disk, solid state disk and flash etc) with SCM. The SJM is put forward to address the inefficiencies of the JBD block device journaling mechanism of Linux. In general, the design and realization process of *SJM* has taken all aspects of the problems into account, but SJM is not perfect. The proposed design utilizes the SCM more as a storage device, compared with the log device layout of JBD, and the layout model of SJM makes full use of the SCM characteristics of byte addressing. As for memory, the logical sequence of transactions can be fully realized by the list, without ensuring the physical order of the log labels. In future work, SJM can be applied to a storage system based on SCM and further improvements can be made to SJM for small file write performance.

6. ACKNOWLEDGMENTS

We would like to thank Stephen MacKay for commenting on earlier drafts of this paper. This work is supported in part by the National 973 Program of China (2011CB302301), the National Natural Science Foundation of China (61472153), the Fundamental Research Funds for the Central Universities (HUST: 2014QN009), and the Natural Science Foundation of Hubei Province (2015CFB192).

7. REFERENCES

[1] IOzone filesystem benchmark. http://www.iozone.org/, 2006.

[2] F. Chen, M. Mesnier, and S. Hahn. A protected block device for persistent memory. In *Proc. of the 30th International Conference on Massive Storage Systems and Technology (MSST)*, 2014.

[3] J. Chen, Q. Wei, C. Chen, and L. Wu. FSMAC: A file system metadata accelerator with non-volatile memory. In *Proc. of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2013.

[4] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2012.

[5] J. Condit, E. B. Nightingale, C. Frost, E. I. B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 133–146, 2009.

[6] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Proc. of the IEEE 27th International Conference on Data Engineering (ICDE)*, pages 1221–1231, 2011.

[7] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4/5):439–447, 2008.

[8] K. M. Greenan and E. L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proc. of the 6th ACM & IEEE International conference on Embedded software*, pages 178–187, 2006.

[9] K. M. Greenan and E. L. Miller. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proc. of the 3rd workshop on on Hot Topics in System Dependability*, pages 22–38, 2007.

[10] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter Technical Conference (ATC)*, pages 1–6, 1994.

[11] Y. Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, December 2008.

[12] J. Katcher. Postmark: A new filesystem benchmark. http://www.netapp.com/tech_library/3022.html, 1997.

[13] M. Katiyar. Journalling Block Device (JBD). http://www.linuxforums.org/articles/journalling-block-device-jbd-_1544.html, November 2011.

[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proc. of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13. ACM, 2009.

[15] E. Lee, H. Bahn, , and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 73–80, 2013.

[16] E. Lee, J. E. Jang, and T. Kim. On-demand snapshot: An efficient versioning file system for phase-change memory. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2841–2853, 2013.

[17] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn. Shortcut-JFS: A write efficient journaling file system for phase change memory. In *Proc. of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2012.

[18] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Proc. of the 32nd IEEE International Conference on Computer Design (ICCD)*, 2014.

[19] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 1–17, 1999.

[20] H. Reiser. Reiserfs. http://www.namesys.com, 2004.

[21] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proc. of the 12th USENIX conference on File and Storage Technologies (FAST)*, pages 287–293, CA, USA, 2014. USENIX Association Berkeley.

[22] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[23] S. Tweedie. Ext3, journaling filesystem. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[24] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *Proc. of the the Fourth Annual Linux Expo*, pages 1–8, 1998.

[25] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 61–75, 2011.

[26] Wikipedia. Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page, July 2014.

[27] X. Wu and A. L. N. Reddy. SCMFS: A file system for storage class memory. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1498–1503, 2011.