Minimizing Consistency-Control Overhead with Rollback-Recovery for Storage Class Memory

Hyun Ku Lee Junghoon Kim Dong Hyun Kang Young Ik Eom

College of Information and Communication Engineering
Sungkyunkwan University
Suwon, 440-746, Korea
{hyungooo, myhuni20, kkangsu, yieom}@skku.edu

ABSTRACT

To guarantee system reliability, consistency-control scheme, such as journaling, is widely used in modern computing environments. However, existing consistency-control schemes negatively affect system performance because they frequently issue write commands to the storage device in order to guarantee data consistency and durability. Especially, when a storage class memory (SCM) is attached to memory bus as storage device, existing consistency-control schemes should be revisited because the buffer cache layer can be potentially eliminated. In this paper, we propose a novel consistency-control scheme, called MinL2R, which guarantees system reliability in the systems that have SCM device attached to memory bus. MinL2R significantly reduces the amount of backup data by performing *data logging* that records the original data related to updates at the granularity of a byte. In addition, MinL2R supports *pointer logging* in order to further reduce the logging overhead, when the size of updates is larger than the size of half of a block. To evaluate our scheme, we implemented the MinL2R prototype with SCM-aware file systems, such as BPFS and Shortcut-JFS. Then, we compared the performance of MinL2R against different file systems including Ext4, BPFS, and Shortcut-JFS, by using several I/O benchmarks. The experimental results clearly show that MinL2R outperforms by up to 286%, 291%, and 395% on BPFS, Shortcut-JFS, and the Ext4 file system with journal mode, respectively.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management – *Consistency-control scheme*

General Terms

Performance, Design, Reliability

Keywords

Storage Class Memory, File System, Data Consistency, Rollback-recovery

IMCOM (ICUIMC)'15, January 8–10, 2014, Bali, Indonesia

1. INTRODUCTION

Recently, consistency-control schemes, such as journaling or shadow paging, are widely used in file systems because system reliability is one of the most important issues in file system design. In order to guarantee data consistency and durability, almost all file systems store the logged data in the journal area of the storage, before writing new data in original location. However, existing consistency-control schemes can negatively affect the system performance due to both duplicated writes and expensive sync operations. Especially, traditional consistency-control schemes are designed in the buffer cache architecture because legacy-computing systems employ the buffer cache layer to mitigate poor storage performance. However, if SCM device is attached to memory bus, the buffer cache layer can be potentially eliminated because SCM has attractive characteristics, such as high performance, non-volatility, byte-addressability, and unlimited endurance.

Recently, several companies, such as Everspin, have been commercializing the SCM [1-2] to use it as main memory or as storage device. However, SCM device, which is attached to memory bus without the buffer cache layer, cannot be directly used at this time because it has a consistency issue caused by frequent CLFLUSH operations, which flushes all data on CPU cache to the SCM storage. Unsurprisingly, previous studies have proposed various SCM file systems that consider both CPU instructions and SCM characteristics [5-6]. However, they do not consider the consistency-control overhead and it significantly reduces the system performance.

In this paper, we propose a novel consistency-control scheme, called MinL2R (minimal logging to rollback-recovery). MinL2R reduces the consistency-control overhead by performing either *data logging* or *pointer logging* with rollback-recovery, which copies the original data instead of new data before updating them, so that we can undo the update when system failure occurs. Especially, *pointer logging* eliminates the consistency-control overhead significantly by logging block pointers instead of the data in a block.

The key contributions of this work can be summarized as follows:

- We propose a new consistency-control scheme, called MinL2R, which guarantees data consistency and durability in the systems that have SCM device attached to memory bus. MinL2R provides data and pointer logging in order to keep data consistency and thus significantly reduces the amount of writes for system recovery.

- We have implemented the MinL2R prototype with SCM-aware file systems, such as BPFS and Shortcut-JFS, and compared them with several benchmark tools.
- The evaluation results clearly show that MinL2R outperforms by up to 286%, 291%, and 395% on BPFS, Shortcut-JFS, and the Ext4 file system with journal mode, respectively.

The rest of the paper is organized as follows. Section 2 explains the background and motivation of our scheme. Section 3 describes the design of MinL2R in detail, and Section 4 shows the evaluation results of our scheme. Lastly, Section 5 concludes our paper.

2. BACKGROUND & MOTIVATION

In this section, we describe existing consistency-control schemes. Also, we discuss some consistency issues when SCM device is attached to the memory bus in the system.

2.1 Existing Consistency-control Scheme

To guarantee both data consistency and durability, most file systems generally store backup data (i.e., journal and log data) before writing new data to the original location by using their consistency-control scheme. Existing consistency-control schemes can be classified into two categories, journaling and rollback-recovery scheme. Journaling scheme stores new data to the reserved area, such as journaling area, (this operation is called commit) and then overwrites the original data in place (this operation is called checkpoint). In contrast, rollback-recovery scheme stores the original data to the reserved area, such as logging area, before overwriting the original data in place. When system failure occurs during commit stage, both schemes ignore the last write operation because new data is not reflected to the original data at that time. On the other hand, when system failure occurs during checkpoint stage, both schemes recover the last consistent state of the file system by restoring the duplicated data in reserved area. However, existing consistency-control schemes significantly degrade the system performance because they periodically issue write operations to secondary storage during the commit and checkpoint period.

2.2 SCM-aware Consistency-control Scheme

To guarantee high performance and consistency, storage researchers focused on a large-sized SCM storage because SCM have attractive characteristics, such as byte-addressability, non-volatility, fast access latency, and low power consumption. Especially, previous studies exploit the byte-addressability of the SCM storage because a large-sized SCM storage can be directly connected to memory bus, which supports byte-level access. Some studies also propose SCM-based file system with consistency control scheme to take benefits of a large-sized SCM storage. Shortcut-JFS is designed to reduce journaling overhead. In order to minimize the amount of journaling data, Shortcut-JFS performs either differential logging or in-place checkpointing according to the size of logged data. The differential logging first stores new data to journaling area at granularly of byte-level and then overwrites original data instead of an entire block or page by exploiting byte-addressability of the SCM storage. The in-place checkpointing only modifies the pointer indicating the block, which is stored in journaling area at last commit state, instead of traditional checkpoint operations. BPFS [5] proposed Short-Circuit Shadow Paging (SCSP), which is derived from shadow paging scheme. SCSP guarantees data consistency as well as reduces the amount of writes issued to the SCM storage by limiting the update propagation. However, this scheme requires special hardware features such as atomic 8-byte writes and epoch barriers to guarantee data synchronization between CPU cache and the large-sized SCM storage.

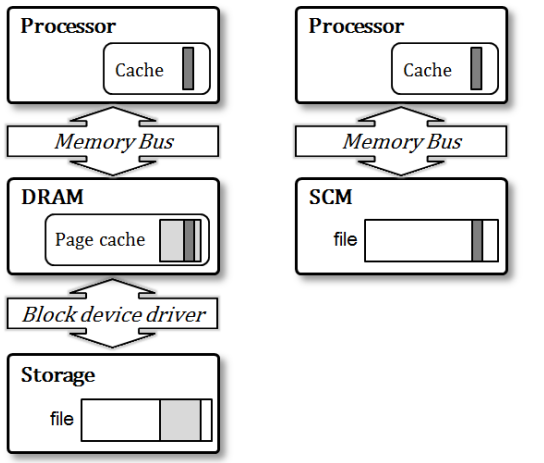

Figure 1. Comparison of system architectures

2.3 Motivation

If a large-sized SCM storage is directly connected to memory bus, existing I/O stacks should be revisited because the SCM storage provides several advantages in the system performance, power consumption, and potential operating cost. Especially, existing page cache layer and consistency control scheme should be redesigned to take benefits of the SCM storage. Fig. 1 shows the traditional architecture, which consists of page cache layer and storage layer, and the large-sized SCM storage architecture. Since performance gap between CPU cache and the large-sized SCM storage can be reduced, the large-sized SCM storage architecture can potentially eliminate page cache layer that is designed to mitigate performance gap between CPU cache and storage. Also, since the SCM storage architecture supports byte-level consistency-control scheme via the memory bus, it can address the problem of existing consistency control scheme, which always performs journaling or logging at the granularity of page even though the page is partially updated. However, in case of this architecture, the consistency issue caused by frequent CLFLUSH operations, which flushes all data on CPU cache to the SCM storage, remains to be addressed. In this paper, we design a novel SCM-aware consistency control scheme, called MinL2R, that minimizes the amount of log data written to the SCM storage by performing either data logging or pointer logging and it guarantees the highest data consistency and durability

3. DESIGN OF MinL2R

3.1 System overview

Fig. 2 shows the overall architecture of the proposed system. A CPU cache is used to reduce the average access time of data from

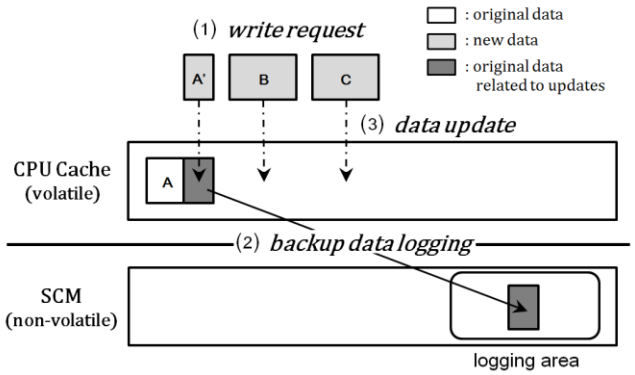

Figure 2. Overall architecture of the proposed system

the SCM storage, which is directly connected to memory bus. The SCM storage is used as main data storage in the proposed system and consists of the file system and logging areas. In the proposed system, the page cache layer, which acts as a transparent cache of storage-backed pages for fast access, is removed because our system provides fast read/write access to the SCM storage through memory bus.

As mentioned previously, ensuring data consistency and durability is the most important factors in designing file system. In the proposed system, file data is transferred between non-volatile SCM storage and volatile CPU cache, and thus consistency-control scheme is required to guarantee file system consistency. In order to reduce consistency-control overhead, MinL2R exploits rollback-recovery scheme, which copies the original data before updating them, so that we can undo the update if system failure occurs. In the example shown in Fig. 2, an application issues a write request, which consists of the modification of block *A* and the insertion of new data of block *B* and *C*. Then, MinL2R logs the original data related to the updates (i.e., overwritten area of block *A*) with the CLFLUSH operation that must be handled synchronously. The CLFLUSH operation flushes cached data in CPU cache to the non-volatile SCM storage in order to guarantee data consistency and durability. In the case of the new data of block *B* and *C*, MinL2R do not need to log original data. Thus, as the size of logged data decreases, consistency-control overhead can be relieved. If MinL2R exploits WAL scheme, which logs the updates to a separate area and periodically writing them to their original locations, the updates *A'*, *B*, and *C* must be logged for ensuring file system consistency, and thus consistency-control overhead can be high. That is why MinL2R exploits rollback-recovery scheme. After logging original data, MinL2R reflects the updates to the file system.

3.2 Write operation

This section describes the write operation of MinL2R in detail. As shown in Fig. 3, the file system manages each file object by using inode, indirect block, and data blocks. The inode data structure is used to represent the file object. Each inode stores file metadata such as file mode, owner UID, file size, last access time, and last data modification time. The indirect block manages all data blocks of the file object. Thus, whenever an application issues a write request to a specific file, both file data and file system metadata including the inode and indirect block should be updated. Before updating them, MinL2R stores their original data, which is related to the updates, with CLFLUSH operation to the logging area, so that we can restore the file system to a consistent state when system failure occurs. This procedure is called COMMIT operation.

In order to further reduce the size of logged data upon COMMIT stage, MinL2R exploits two novel schemes, called *data logging* and *pointer logging*. The data logging scheme logs the original data related to updates and thus is suitable for small updates. On the other hand, the pointer logging scheme logs pointer to the original block, instead of original data, and thus is suitable for large size updates. MinL2R calculates modified data ratio of each block as follows.

$$Modified\ data\ ratio = \frac{size\ of\ original\ changed\ data}{size\ of\ original\ data} \quad (1)$$

If the modified data ratio of a block is smaller than the threshold, MinL2R performs the data logging scheme. On the other hand, when the modified data ratio of the block is larger than or equal to the threshold, MinL2R performs the pointer logging scheme. The optimal threshold for determining logging policy will be further discussed in Section 4.2. After logging process, MinL2R reflects the updates to the file system.

3.3 Read operation

In the traditional file system, an application reads the requested data from the page cache layer. However, the proposed system eliminates the page cache layer to improve system performance. If consistency-control scheme is designed based on WAL scheme, the file system reflects the updates from the logging area to the file system area, whenever read request is issued. However, Since MinL2R reflects the updates to the file system area directly with rollback-recovery scheme, the application can read the requested data from file system area of SCM storage, regardless of logged data. Thus, the proposed system guarantees fast read access time of requested data.

3.4 Checkpoint operation

Whenever a certain threshold size of logging area is filled with logged data, CHECKPOINT operation is required in order to issue new write requests. Upon CHECKPOINT operation, MinL2R flushes the updates from the volatile CPU cache to the non-volatile SCM storage to guarantee data consistency and durability. Then, the used space of logging area can be recycled after the CHECKPOINT operation is completed. In our MinL2R prototype, CHECKPOINT operation is triggered when a fourth of logging area is filled or 5 seconds have passed after the last CHECKPOINT operation.

3.5 Failure recovery

In order to guarantee data consistency and durability, MinL2R logs the backup data before updating them. If the system failure occurs during write operation, MinL2R restores the file system to a consistent state upon system reboot. Depending on the status of the transaction, the failure recovery process is determined. When the status is BEGIN or LOGGING, MinL2R does nothing because the file system does not reflect the updates yet, and thus

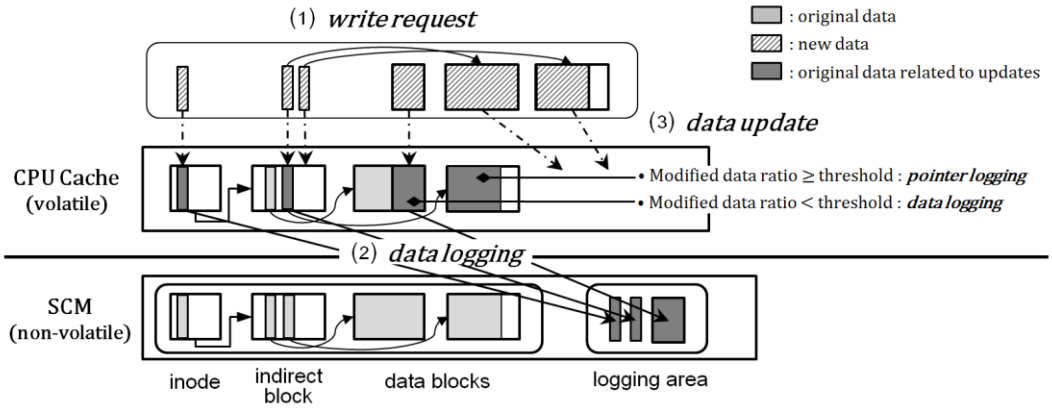

Figure 3. Overall sequence of write operation for MinL2R

guarantees the consistent state. If the status is COMMIT or WRITING, the file system can be partially updated, and thus failure recovery is required. Our failure recovery process is illustrated in Fig. 4. During failure recovery process, MinL2R restores crushed file depending on the type of logged data. If the logged data is made from data logging scheme, MinL2R overwrites the logged data to the original location in order to restore the previous consistent state. If the logged data is made from pointer logging scheme, MinL2R frees the allocated new block and overwrites the logged block pointer to the indirect block in order to restore the previous consistent state. Through this procedure, MinL2R can recover the file system in a short time.

4. EVALUATION

For evaluation, we implemented the MinL2R prototype with SCM-aware file systems, such as BPFS and Shortcut-JFS. Then, we compared the performance of MinL2R against several different file systems including Ext4, BPFS, and Shortcut-JFS.

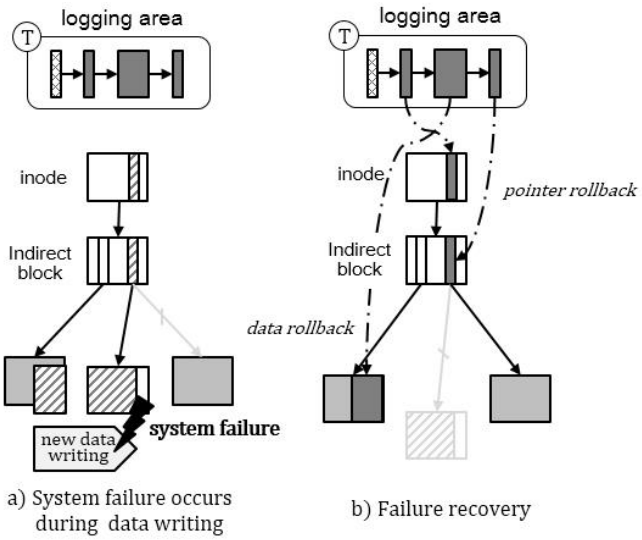

Figure 4. Failure recovery process of MinL2R

4.1 Experimental Setup

We evaluated the performance of MinL2R on a system with 3.4GHz Intel Core i7 CPU, 8GB of main memory. Also, we used DRAM as a large-sized SCM storage because of two reasons. First, a large-sized SCM storage is not commercially available at this time. Second, latency of SCM storage is expected to take similar latency to DRAM. The MinL2R prototype is implemented with ramfs on Linux 3.13.0 kernel. MinL2R extends traditional ramfs to guarantee system consistency, because ramfs does not provide consistency-control scheme. For fair comparison, we implemented BPFS and Shortcut-JFS based on ramfs. Especially, since BPFS requires specific hardware supports (i.e., 8-byte atomic instruction and epoch barrier) to guarantee the system consistency we emulated its consistency-control scheme by using CLFLUSH operation. We also eliminated page cache layer in Shortcut-JFS because MinL2R is designed without page cache. To assess the effectiveness of MinL2R with existing file systems, we mounted the Ext4 with ordered and journal modes on the 1GB RAMdisk, which employs main memory as a storage. We performed our experiments with two benchmarks, Flexible I/O (FIO) and IOZONE benchmark. FIO is commonly used to compare the performance of random I/O [10] and IOZONE is widely used for measuring the file system performance in terms of sequential and random read/writes [9]. To avoid influence of previous experiments, we also rebooted the system whenever a file system is changed.

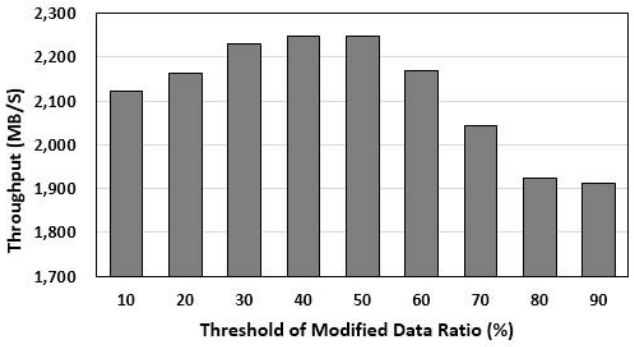

Figure 5. Write throughputs of FIO benchmark by several different threshold of modified data ratio.

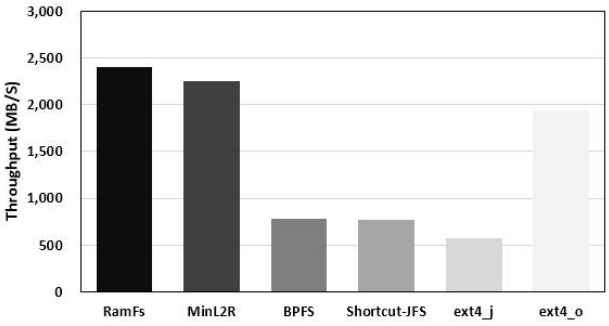

Figure 6. Write throughput comparisons on FIO benchmark

4.2 Optimal threshold for MinL2R

As mentioned Section 3, MinL2R performs either data logging or pointer logging according to pre-defined threshold. If the modified data ratio is smaller than the pre-defined threshold, MinL2R performs *data logging* that stores original data related to updates at the granularly byte-level and then original data is overwritten in-place with new data. Otherwise, MinL2R performs *pointer logging* that stores a pointer, which points to the original data block and then it is overwritten in-place with the new pointer to the data block containing new data (see Fig. 3).

In order to evaluate the effectiveness of pre-defined threshold, we performed experiments varying the threshold sizes. As shown in Fig. 5, MinL2R has the highest write throughput when pre-defined threshold is 50%. As expected, when pre-defined threshold is higher than 50%, write throughput is degraded. The major reason is that the amount of SCM writes increases by storing both large-sized original data and new data. Also, when pre-defined threshold is lower than 50%, the write throughput is degraded because a new data block is frequently allocated for pointer logging. Based on these experimental results, we set the pre-defined threshold to 50%.

4.3 Experimental Results

To evaluate the system performance, we first compared the throughput by using FIO benchmark [10]. We ran FIO benchmark by varying the record sizes from 1k to 32k. As shown in Fig. 6, MinL2R outperforms by up to 286%, 291%, and 395% on BPFS, Shortcut-JFS, and the Ext4 file system with journal mode, respectively. Especially, MinL2R improves the throughput by up to 115% compared to the ext4 file system with *ordered* mode even though MinL2R guarantees the highest consistency. We also measured the throughput with IOZONE benchmark [9], including sequential write, random write, and fwrite, varying the record size ranging from 1KB to 512KB. Fig. 7 shows the throughput of each file system. As shown in Fig.7, MinL2R outperforms other file systems in all record sizes except ramfs. Specifically, MinL2R significantly improves the throughput by up to 750% compared to BPFS that employs CLFLUSH operation to emulate hardware based consistency-control scheme, such as 8-byte atomic instruction and epoch barrier. In addition, as record size is higher, throughput gap between MinL2R and BPFS increases. In order to analyze these throughput gaps between MinL2R and BPFS in detail, we calculated the size of backup data during the execution of the IOZONE benchmark. Fig. 8 shows the amount of data written with logging for MinL2R and BPFS. Fig. 8 clearly shows that MinL2R significantly reduces the amount of data written with

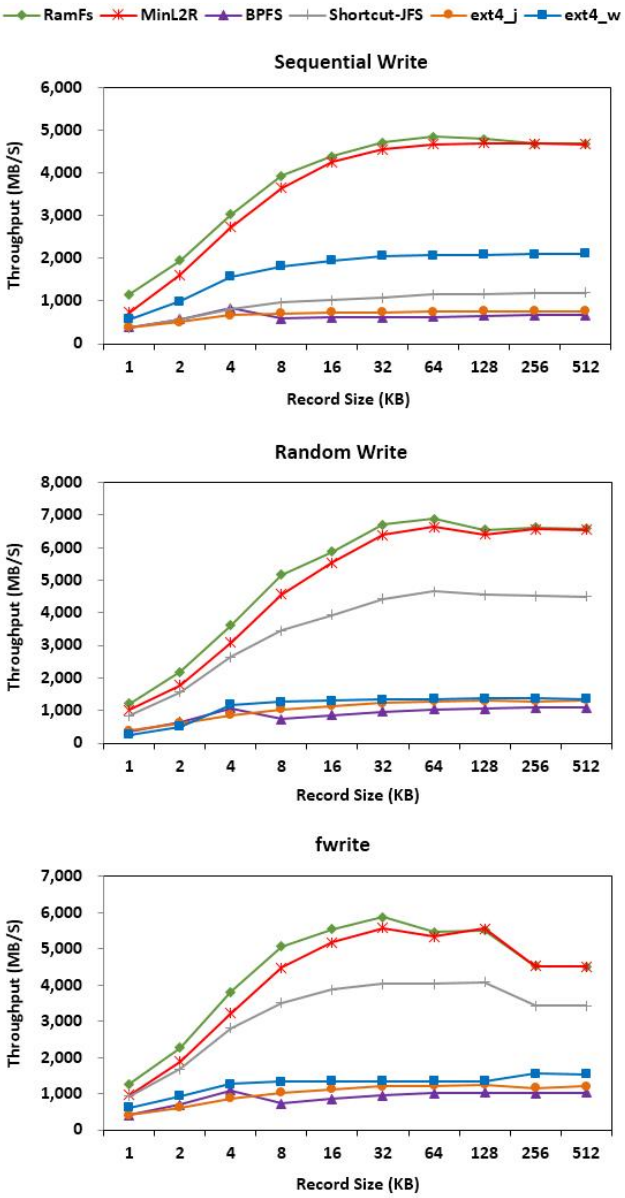

Figure 7. Write throughput comparisons on IOZONE benchmark

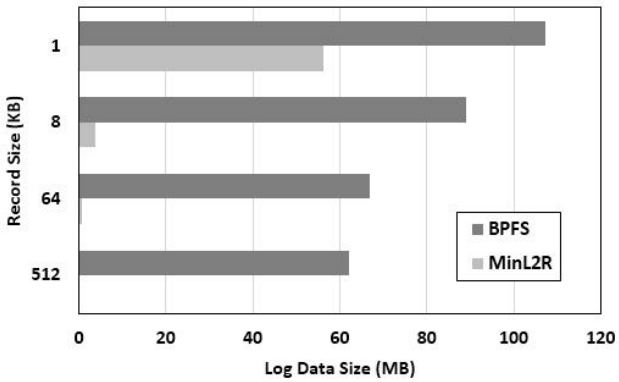

Figure 8. Backup data size comparison between BPFS and MinL2R

logging. This is because MinL2R can perform pointer logging when modified block ratio is higher than pre-defined threshold. As expected, Fig.6 and Fig. 7 show that ramfs outperforms MinL2R in all configurations. However, ramfs does not guarantee data consistency because it never provides any consistency-control scheme.

5. CONCLUSION

In this paper, we proposed a novel consistency-control scheme for the systems that have SCM storage, called MinL2R. We designed MinL2R considering the characteristics of SCM device carefully. Thus, we can significantly reduce the amount of writes for system recovery by adapting rollback-recovery technique. Furthermore, we eliminate unnecessary copying overhead by choosing the suitable logging scheme depending on the modified data ratio. Experimental results show that MinL2R greatly improves the overall I/O performance without any loss of data consistency. For future work, we plan to explore the optimal file system structure for the systems that have SCM storage.

6. ACKNOWLEDGMENTS

This work was supported by ICT R&D program of MSIP/IITP. [10041244, SmartTV 2.0 Software Platform]

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730)

7. REFERENCES

[1] Everspin's CEO explains the company's technology and business. http://www.mram-info.com/everspins-ceo-explains-companys-technology-and-business, 2014.

[2] Everspin: 256Mb ST-MRAM chips coming soon. http://www.mram-info.com/everspin-256-mbit-st-mram-chips-coming-soon, 2014.

[3] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, Vol. 52, No. 4.5, pages 439-447, 2008.

[4] E. Lee, D. Jin, K. Koh, and H. Bahn. Is buffer cache still effective for high speed pcm (phase change memory) storage?. In *Proceedings of the Parallel and Distributed Systems (ICPADS)*, pages 356-363, 2011.

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

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

[7] X. Wu and N. Reddy. Scmfs: A the file system for storage class memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, No. 39, pages 1–11, 2011.

[8] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91-104, 2011.

[9] D. Norcott. Iozone filesystem benchmark. http://www.iozone.org/docs/IOzone_msword_98.pdf.

[10] J. Axboe. FIO (Flexible IO Tester). http://git.kernel.dk/?p=fio.git;a=summary, 2014.