International Journal of Advanced Smart Convergence Vol.12 No.1 76-81 (2023)
http://dx.doi.org/10.7236/IJASC.2023.12.1.76

IJASC 23-1-10

Design of an Efficient In-Memory Journaling File System for Non-Volatile Memory Media

Hyokyung Bahn

Professor, Department of Computer Engineering, Ewha University, Professor, Korea
bahn@ewha.ac.kr

Abstract

Journaling file systems are widely used to keep file systems in a consistent state against crash situations. As traditional journaling file systems are designed for block I/O devices like hard disks, they are not efficient for emerging byte-addressable NVM (non-volatile memory) media. In this article, we present a new in-memory journaling file system for NVM that is different from traditional journaling file systems in two respects. First, our file system journals only modified portions of metadata instead of whole blocks based on the byte-addressable I/O feature of NVM. Second, our file system bypasses the heavy software I/O stack while journaling by making use of an in-memory file system interface. Measurement studies using the IOzone benchmark show that the proposed file system performs 64.7% better than Ext4 on average.

Keywords: *File System, Non-Volatile Memory, Reliability, Journaling, Ordered Mode.*

1. Introduction

Reliability is one of the important issues in the design of file systems. In particular, sudden power failures can happen in systems like mobile devices and data recovery is necessary when system crashes occur. To cope with this situation, journaling has been widely adopted in reliable file systems such as Ext4 [1]. Journaling writes updated data to a temporary storage location called the journal area first, and then reflects them to the persistent location of the file system later. Writing updated data to the journal area first protects data from being corrupted when a sudden system crash occurs because a consistent version of the data always exists either in the journal area or in the persistent file system location. In contrast, corrupted data may not be recovered if we flush updated data directly to its persistent location since the data can be partially updated during the power failure situation [2, 3].

In this article, we present a new journaling file system designed specifically for NVM (non-volatile memory) devices. NVM is a byte-addressable medium like DRAM, so traditional journaling techniques designed for block devices are not appropriate for NVM in two respects. First, if we adopt traditional journaling as it is in NVM, an entire block should be journaled even though a single byte has been modified. As metadata journaling

Manuscript Received: January. 8, 2023 / Revised: January. 11, 2023 / Accepted: January. 13, 2023
Corresponding Author: bahn@ewha.ac.kr
Tel: +82-2-3277-2368, Fax: +82-2-3277-2306
Professor, Department of Computer Engineering, Ewha University, Professor, Korea

Copyright© 2023 by The Institute of Internet, Broadcasting and Communication. This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0)

is the default option of most file systems, which involves journaling of just a few bytes, it may cause significant inefficiency to NVM. Second, when an I/O request is issued on a traditional block storage device, the request traverses through a series of software stack layers to access the storage device. In NVM-based storage systems, as the access latency is very fast, the relative impact of this software stack overhead increases dramatically. A recent study reports that the software stack overhead is responsible for less than 1% of the storage access latency under the hard disk storage, but it is responsible for over 90% in NVM storage when the same software I/O stack is used [4]. Therefore, it is important to alleviate this overhead when designing file systems for NVM storage.

Motivated by this, we design and implement a journaling file system that can reduce the amount of data journaled and the software stack overhead significantly when the underlying storage device is NVM. Previous studies on NVM usually adopt NVM as a block device that needs to pass through the software I/O stack, and also do not exploit the byte-addressable features of NVM [5, 6, 7]. Unlike previous studies, we devise an efficient journaling mechanism by the byte-addressable I/O feature of NVM. Specifically, during the journaling process, our file system flushes only the modified part of metadata to the byte-addressable journal area in NVM, while existing journaling file systems perform I/O for a whole block [1]. Also, our file system need not pass through heavy software I/O stacks during journaling by making use of the in-memory file system interface. Measurement studies with IOzone benchmarks for the four representative I/O scenarios show that the proposed file system performs better than Ext4 by 64.7% on average.

2. Related Work

To buffer the speed gap between main memory and storage, operating systems adopt the buffer cache layer that holds file data requested in a certain memory space. By utilizing this buffer cache, file systems such as Ext4 do not need to access slow storage when the same data in the buffer cache is subsequently requested [8]. However, as the buffer cache uses volatile DRAM, if the system crashes while writing modifications to storage, the file system may be inconsistent or even corrupted. To cope with this situation, reliable file systems such as Ext4 adopt the journaling mechanism [1, 3]. Instead of writing modified data directly to its persistent location in the file system, journaling writes modifications to the journal area first and then reflects them to the persistent location later. By so doing, journaling guarantees file system consistency in any crash situations.

Journaling groups data to be updated atomically and manages them as a single transaction. To implement journaling semantic, three types of transaction lists are used in journaling file systems [1, 8]. The first is a running transaction list that maintains blocks that become dirty after the completion of the previous journaling. When journaling starts, the running transaction list is converted to a commit transaction list, and a new running transaction list is created to insert updated blocks for the next journaling period. During the journaling, all dirty blocks within the transaction list are journaled together. When all blocks in this transaction is successfully written to the journal area, a commit mark is placed on the journal area. This mark indicates that the transaction will be reflected to the persistent locations in storage even in case of a system crash. After the completion of this journaling, the commit transaction list is converted to the checkpoint transaction list, which is maintained until the journal data are reflected to their persistent file system locations through checkpointing. Unlike journaling operations that are performed frequently to protect data from being lost (e.g., default period is 5 seconds in Ext4), checkpointing intervals are relatively long (e.g., 5 minutes).

Journaling provides reliable file system states, but it generates additional storage writing. As a compromise between performance and reliability, journaling file systems offer different journaling levels such as metadata journaling and full data journaling. In metadata journaling, only metadata is journaled and regular data blocks

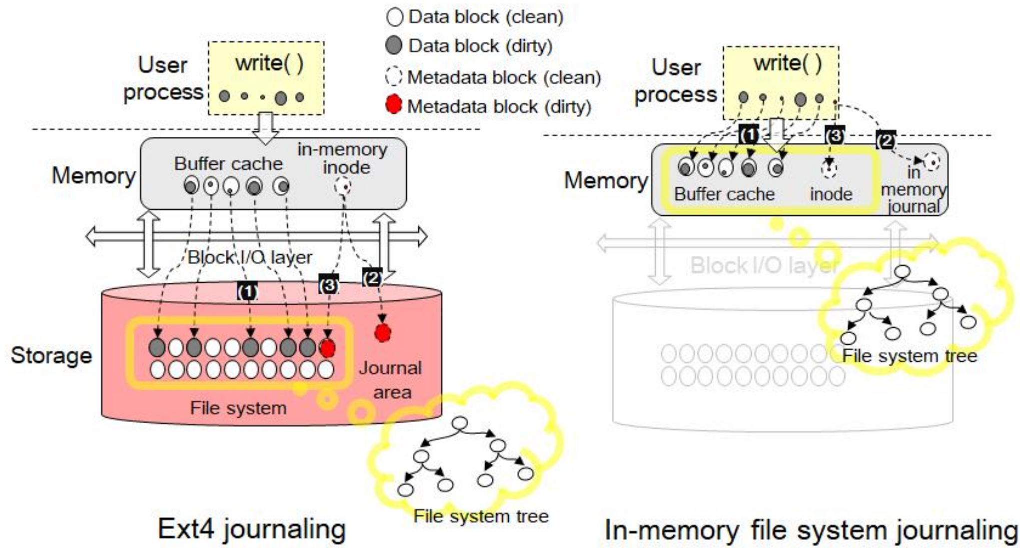

Figure 1. Comparison of journaling in Ext4 and the proposed in-memory file system.

are directly flushed to their persistent file system locations. So, the file system structure is protected while file contents can be corrupted in metadata journaling. In full data journaling, the file system journals both metadata and regular data. This guarantees the complete consistency of a file system, but incurs significant overhead due to writing the same data to storage twice. To compromise between reliability and performance, some file systems like Ext4 provide a kind of metadata journaling called the ordered-mode journaling where only the metadata is journaled but regular data should be flushed before journaling metadata in order to minimize the chance of data corruption such as generating dangling pointers [9]. Journaling is basically performed periodically but it can also be triggered by synchronous writes from user processes.

3. In-Memory Journaling File System for NVM

The file system proposed in this article is designed to utilize the byte-accessibility of NVM by making use of an in-memory file system interface. Thus, unlike traditional file systems, a certain memory address space is mapped directly into the file system tree, thereby bypassing the buffer cache layer as shown in Figure 1. This file system structure reduces unnecessary memory copy operations and eliminates software stack overhead during I/O operations. It also eliminates the inefficiencies of traditional file systems where storage is accessed by block size. However, if a power failure occurs while data is being written to the file system, the data may become inconsistent.

To resolve this problem, journaling techniques are widely used, but it is not possible to use journaling designed for block devices as it is in an in-memory file system. In this article, we aim to design a journaling technique for an in-memory file system that provides the same level of data reliability as the journaling used in existing file systems like Ext4. To this end, we analyze the source code of journaling block device (JBD) used in Linux operating systems and port it to the journaling module of our in-memory file system.

Meanwhile, as the overhead of full data journaling is large, most journaling file systems including Ext4 journal only metadata as a default. Similarly, we use the ordered mode, which is a widely used journaling option that journals only metadata, in our in-memory file system design. This has the advantage of providing the same level of reliability as the existing file systems while maximizing the efficiency of the in-memory file

system.

When a user process makes a write system call, our file system writes directly to the file system location in memory without passing through the buffer cache. This is different from traditional file systems, which write to buffer cache first and then reflect the writes to the file system location. Therefore, in order to guarantee reliability in an in-memory file system, journaling should be performed for every write request from a user process, which causes enormous overhead. Although the software I/O stack overhead can be eliminated, journaling in in-memory file systems requires write-twice behaviors for every write system call. This causes excessive overhead compared to periodic flushing performed in existing journaling file systems. However, if the journaling option is set to the ordered mode, most of this overhead can be removed as the targets of journaling are limited to inode parts. Also, our in-memory journaling supports atomicity of each write request, which is different from traditional journaling that supports periodic transactions.

Let us describe in detail how our file system works. When a write request comes from a user process, regular data is first written directly to the corresponding data location in the file system. Unlike conventional file systems, writing is performed in byte units for the modified file location similar to memory write operations, which is very efficient because it does not pass through the software I/O stack. These write operations correspond to the same amount of work as write operations to the buffer cache in traditional file systems. When writing of regular data is finished, inode journaling is performed in a specific memory area. This journaling is completed by recording a COMMIT mark. A write operation that has been committed means a state in which the file system can be restored even if a system crash occurs. After journaling for an inode, the inode is flushed to a persistent location in the file system. Since all of these steps consist of only memory access operations, they incur light overhead similar to writing to the buffer cache in traditional file systems.

In summary, the proposed file system has the advantage of eliminating the overhead of traditional journaling in terms of the I/O paths to be traversed and the I/O size to be journaled.

4. Experimental Results

To validate the effectiveness of the proposed file system, we compare the performance of our file system that we call NVMFS in comparison with Ext4 through measurement experiments. There are other journaling file systems such as ReiserFS, but we chose Ext4 as it is the most widely used in systems ranging from mobile devices to high-end servers. Our experimental platform consists of an Intel Core i5-3570 CPU running at 3.4GHz, 8GB of DDR3-1600 memory and 1TB of SATA 3Gb/s HDD. We perform experiments on Linux kernel 3.10.20. We emulate the performance of NVM by making use of DRAM on DIMM slots. This is the upper-bound performance of NVM as STT-MRAM is expected to provide nearly identical read/write performances of DRAM [10].

In the Ext4 file system case, though NVM is put on DIMM slots, it is recognized as a block device as the Ramdisk device driver is installed on it. For Ext4's journaling, we use the ordered mode as is the default configuration in Ext4. Our file system also performs the ordered mode journaling, but it makes use of an in-memory file system interface, thereby accessing NVM via byte-addressable interfaces. As the two file systems equally use the ordered mode, they guarantee the same reliability level. We measure the performance of systems using the IOzone benchmark that assesses the performance of storage systems by generating a series of specific access patterns [11]. We use four IOzone scenarios: random read, random write, sequential read, and sequential write, which are the most popular scenarios used in many other studies [3, 4]. Following the

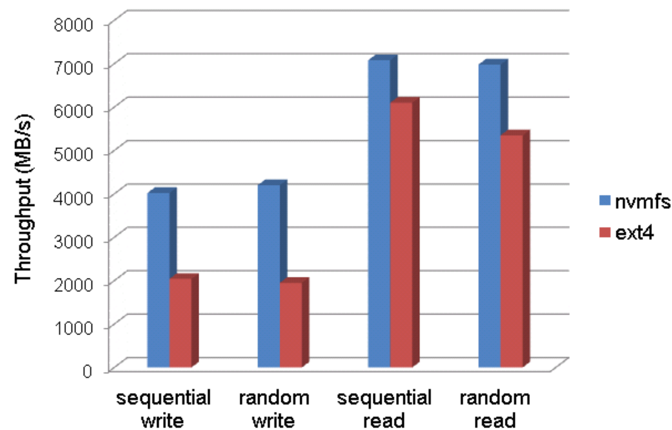

Figure 2. Comparison of the proposed file system with Ext4 with IOzone benchmarks.

default configuration of IOzone, the operation unit ranges from 4KB to 16MB. The total footprint of the benchmark is set to 2GB.

Figure 2 shows the throughput of the proposed file system in comparison with Ext4. As shown in the figure, our file system performs better than Ext4 in all cases of IOzone scenarios. Specifically, when we evaluate the throughput difference between the two file systems divided by the throughput of Ext4, the performance improvement of our file system against Ext4 by is 64.7% on average. This is because our file system reduces I/O traffic for small-sized requests and also improves I/O efficiency by eliminating the software I/O stack overhead. Specifically, our in-memory file system uses a certain part of the buffer cache in the kernel address space as the file system location and thus, it does not use the buffer cache during I/O operations. This allows our file system to reduce the synchronization cost between memory and storage. In contrast, Ext4 should pass through the buffer cache layer and the software I/O stack to reach NVM storage incurring considerable overhead.

When comparing the results of read and write, it can be seen that the performance improvement of our file system is relatively small in read operations. This is because a large portion of read requests are absorbed in the buffer cache layer without actual storage I/Os. In write operations, our file system shows significant performance improvement compared to Ext4, where periodic storage writing is essential to maintain consistency even for data that already exists in the buffer cache, whereas our file system does not have such operations.

5. Conclusion

Journaling techniques are widely adopted in reliable file systems such as Ext4 to resist sudden power failure situations. However, we showed that traditional journaling is not appropriate for byte-addressable NVM in two respects. First, traditional journaling should be performed by an entire block size even though a single byte has been modified. Second, traditional journaling depends on I/O operations that should pass through a series of software stack, which incurs large overhead in NVM. Based on these observations, we presented a new journaling file system designed appropriately for NVM. Specifically, our journaling file system takes advantage of the byte-addressable feature of NVM and is designed based on in-memory file system interfaces,

thereby allowing byte-level journaling without passing through software I/O stacks. Measurement studies with IOzone benchmarks showed that the proposed file system performs better than Ext4 by 64.7% on average.

Acknowledgement

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C1009275) and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-02068, Artificial Intelligence Innovation Hub).

References

[1] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," Proc. USENIX Annual Technical Conf. (ATC), 2005.

[2] Y. Choi and S. Ahn, "Separating the file system journal to reduce write amplification of garbage collection on ZNS SSDs," Journal of Multimed. Inf. Syst., vol. 9, no. 4, pp. 261-268, 2022.
DOI: https://doi.org/10.33851/JMIS.2022.9.4.261

[3] E. Lee, S. Hoon Yoo and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," IEEE Trans. Comput. (TC), vol. 64, no. 5, pp. 1349-1360, 2015.
DOI: https://doi.org/10.1109/TC.2014.2329674.

[4] E. Lee, H. Bahn, S. Yoo and S. H. Noh, "Empirical study of NVM storage: an operating system's perspective and implications," Proc. IEEE MASCOTS Conf., pp. 405-410, 2014.
DOI: https://doi.org/10.1109/MASCOTS.2014.56.

[5] T. Cai, Y. Ma, P. Liu, D. Niu, and L. Li, "A new NVM device driver for IoT time series database," Micromachines, vol. 13, no. 3, article 385, 2022.
DOI: https://doi.org/10.3390/mi13030385

[6] D. Kim, E. Lee, S. Ahn, H. Bahn, "Improving the storage performance of smartphones through journaling in non-volatile memory," IEEE Trans. Consum. Electron. (TCE), vol. 59, no. 3, pp. 556-561, 2013.
DOI: https://doi.org/10.1109/TCE.2013.6626238

[7] Y. Park and H. Bahn, "Modeling and analysis of the page sizing problem for NVM storage in virtualized systems," IEEE Access, vol. 9, pp. 52839-52850, 2021.
DOI: https://doi.org/10.1109/ACCESS.2021.3069966

[8] E. Lee, H. Bahn, and S. Noh, "A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory," ACM Trans. Storage, vol. 10, no. 1, pp. 1-14, 2014.
DOI: https://doi.org/10.1145/2560010

[9] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," Proc. Linux Symp., vol. 2, pp. 21-33. 2007.

[10] W. Zhao, J. Xu, X. Wei, B. Wu, C. Wang, W. Zhu, W. Tong, D. Feng, and J. Liu, "A low-latency and high-endurance MLC STT-MRAM-based cache system," IEEE Trans. CAD, vol. 42, no. 1, pp. 122-135, 2023.
DOI: https://doi.org/10.1109/TCAD.2022.3169458

[11] W. Norcutt, the IOzone Filesystem Benchmark, http://www.iozone.org/.