Write Optimization of Log-structured Flash File System for Parallel I/O on Manycore Servers

Chang-Gyu Lee* Hyunki Byun* Sunghyun Noh, Hyeongu Kang, Youngjae Kim†
Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
{changgyu,bhyunki,nsh0249,hyeongu,youkim}@sogang.ac.kr

ABSTRACT

In Manycore server environment, we observe the performance degradation in parallel writes and identify the causes as follows – (i) When multiple threads write to a single file simultaneously, the current POSIX-based F2FS file system does not allow this parallel write even though ranges are distinct where threads are writing. (ii) The high processing time of *Fsync* at file system layer degrades the I/O throughput as multiple threads call *Fsync* simultaneously. (iii) The file system periodically checkpoints to recover from system crashes. All incoming I/O requests are blocked while the checkpoint is running, which significantly degrades overall file system performance. To solve these problems, first, we propose file systems to employ a fine-grained file-level *Range Lock* that allows multiple threads to write on mutually exclusive ranges of files rather than the course-grained `inode mutex` lock. Second, we propose *NVM Node Logging* that uses NVM as an extended storage space to store file metadata and file system metadata at high speed during *Fsync* and checkpoint operations. In particular, the *NVM Node Logging* consists of (i) a fine-grained inode structure to solve the write amplification problem caused by flushing the file metadata in block units and (ii) a Pin Point NAT (Node Address Table) Update, which can allow flushing only modified NAT entries. We implemented *Range Lock* and *NVM Node Logging* for F2FS in Linux kernel 4.14.11. Our extensive evaluation at two different types of servers (single socket 10 cores CPU server, multi-socket 120 cores NUMA CPU server) shows significant write throughput improvements in both real and synthetic workloads.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '19, June 3–5, 2019, Haifa, Israel
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325828

CCS CONCEPTS

• **Software and its engineering → File systems management**;

KEYWORDS

Manycore OS, File System, Non-volatile Memory

1 INTRODUCTION

Data-intensive applications such as relational and NoSQL databases perform parallel I/O for high I/O throughputs [1–6]. Parallel I/O operations allow multiple threads to read and write to a single file concurrently, thus increasing the I/O throughput of applications by exploiting full parallelism of the underlying storage system. For example, a typical relational database such as MySQL uses InnoDB engine to store table data. The InnoDB engine stores a table in a file using a B+ tree. If the table is a single file, then parallel I/Os concurrently perform write or update queries on the same table. In addition, HPC scientific applications perform periodic checkpointing to resume execution after recovery in case of failures [7]. In particular, N-to-1 checkpointing maximizes I/O performance by using parallel I/O, which allows multiple MPI processes to write their own intermediate results concurrently to a single file on mutually exclusive regions [8, 9].

Manycore servers have hundreds of cores on a single machine and provide massive parallelism [10, 11]. With the advancement in hardware technologies, the application level parallelism is becoming more prominent. The parallel I/O of applications running on such servers can expect higher performance with many computing cores. High performance database engines are designed to handle queries concurrently by utilizing this high computational parallelism of the Manycore server [1, 12–14]. The I/O performance of such applications highly depends on the parallel write performance of the storage device. The parallel reads are less susceptible to storage performance because they can benefit from low memory latency through memory caching. On the other hand, the processing time of parallel writes is directly affected by the

*Both authors contributed equally to this work.
†Y. Kim is a corresponding author.

performance of hardware storage devices, which is much slower than main memory.

The most intuitive way to effectively address the performance issues of parallel writes mentioned above is to use high-performance storage device with high parallelism. PCIe-based high-performance SSDs use multiple read/write I/O queues to provide high I/O parallelism and bandwidth at the same time [15]. However, simply employing a high-performance SSD does not necessarily improve the parallel write I/O performance of the application. The application needs to access the SSD storage device, which is a block interface, with the help of the file system such as Ext4 [16], Btrfs [17], F2FS [18] of the OS. Another alternative is to optimize I/O middleware such as DB engine or I/O library in the application. However, in this study, we focus on parallel write optimizations within the OS file system on which I/O middleware depends. We reveal that the file system on a Manycore server can be a bottleneck to prevent I/O performance from improving when performing parallel writes.

F2FS [18] is a Linux native file system optimized for SSD. F2FS employs a log structured design to compensate for the poor performance of random write patterns in NAND Flash-based SSDs. However, when multiple threads write to a single file in parallel, the Linux VFS layer allows multiple writes, but the implementation of the native file system F2FS does not allow concurrent writes. This is because the `inode mutex` lock prevents other writes from accessing the file even if each write accesses another offset of the file. In addition, applications such as databases use *Fsync* system call to persist file data and file metadata to storage devices. However, the *Fsync* system call has a high latency because it must perform block I/O to send the file data and the file metadata in memory to the storage device. In addition, F2FS performs checkpointing when the amount of file data in memory reaches a certain threshold. However, during F2FS checkpointing, all write requests are blocked in order to guarantee consistency of the file system, resulting in a sudden drop in file system performance.

This paper has the following contributions:

- When multiple threads execute parallel writes to one file, they can not be processed simultaneously by the file system, even if their writing areas (file offsets) on the file do not overlap. To allow parallel writes to a single file, we propose to implement a fine-grained file-level *Range Lock* rather than a coarse-grained `inode mutex` lock in F2FS, allowing multiple threads to write to different offsets in a single file, eventually improving parallel write performance.
- *Fsync* is a system call to store file data and file metadata in storage. The speed of storage greatly affects the latency of *Fsync* system calls. Further, frequent *Fsync* calls cause file system checkpointing. All incoming write I/Os are blocked during the checkpointing, resulting in a dramatic drop in file system I/O performance. For high-speed *Fsync* call handling and checkpointing, we propose *NVM Node Logging* to use Non-volatile Memory (NVM) as an extension of storage to store file metadata and file system metadata at high speed.
- When calling *Fsync*, a write amplification problem may occur. The SSD is a block device with a single I/O write unit of 4 KB. If the data contents changed are smaller than 4 KB, then the write bandwidth may be lost due to write amplification. We take into consideration the characteristics of byte-addressability of NVM, so that only modified data can be written in NVM area. For this, we propose a Pin Point NAT Update mechanism, which solves the problem of wasted bandwidth due to write amplification from the existing SSD block interface.
- We have implemented both file-level *Range Lock* and *NVM Node Logging* by modifying the F2FS file system for evaluation. Both ideas were implemented on F2FS and evaluated using 10 cores and 120 cores Manycore servers with both synthetic and realistic workloads. In particular, in a Manycore server evaluation of 120 cores, file-level *Range Lock* implementation showed a 14.9× performance improvement over native F2FS performance and *NVM Node Logging* showed a 35.8% increase in throughput over F2FS. In the Filebench experiment, F2FS, which implemented both file-level *Range Lock* and *NVM Node Logging*, throughput increased by 9% compared to native F2FS, reducing tail latency by 71%.

2 BACKGROUND AND MOTIVATION

2.1 SSD Optimized File Systems

NAND Flash-based SSDs offer a number of benefits over conventional HDDs: improved I/O access time, less power consumption, better resilience to operating in harsh environments with external shocks and hotter temperatures, and lighter-weight devices. But, NAND Flash memory is unable to directly overwrite a storage location in the same manner as magnetic media. Once a location has been written, NAND Flash requires an erase operation before data in that location can be changed. Further complicating matters, read and write operations can be performed at a finer granularity than erase operations. As erase operations are much more expensive in terms of time than read/write operations, SSDs incorporate software to allow out-of-place update operations, which eventually requires a sweep of the storage area to find stale data and consolidate active pages in order to create free blocks. This process, known as garbage collection (GC), can block incoming requests that are mapped to flash chips currently performing erase operations [19]. The frequency

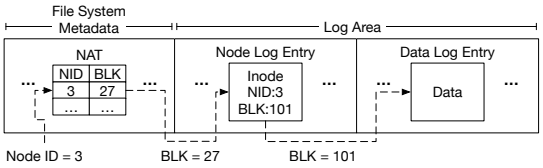


Figure 1: F2FS On-Disk Data Structure for file system metadata and file data and metadata [18].

and overhead of GC operations are significantly affected by random writes and updates with small request sizes.

F2FS [18] is a log-structured file system designed to overcome the aforementioned problems of the NAND Flash-based SSD. F2FS minimizes the GC overhead by sufficiently utilizing SSD's internal parallelism through multi-head logging. Figure 1 shows on-disk layout of F2FS. F2FS on-disk layout can be broadly divided into two parts; The data log section that stores file system metadata and user data, and Node log section that stores file metadata. F2FS manages inode, direct node, and indirect node as Node log, and each Node log's entry is assigned a node id. F2FS uses Node Address Table (NAT), which resides in the file system metadata region on the storage device, to convert the node id to node log entry's block address. In Figure 1, a node which has node id 3 can be accessed with block address 27 via NAT. Consequently, file metadata conversion also conveys the change in NAT.

F2FS uses checkpointing to ensure file system failure consistency. Specifically, it flushes dirty data and metadata, resident in the memory, and NAT metadata into the storage device. However, the problem of checkpointing comes from blocking all incoming write requests during the checkpointing. Due to this blocking problem, resulting from checkpointing, file system throughput can drop drastically and latency increases. The *Fsync* system call can execute file system checkpointing. However, F2FS does not perform checkpointing every time the *Fsync* is called. Instead, write the changes to the journal in the *Fsync* system call. When the journal is full, the journal contents are synchronized with the storage device. Database applications perform many *Fsync* system calls, which is a major cause of performance degradation.

2.2 Non-Volatile Memory Technology

Non-Volatile Memory (NVM) technologies such as PCM [20] and STT-MRAM [21] are the state-of-the-art memory technology which allows DRAM like fast access speed, byte-addressability, and persistency of storage device [22]. Due to its characteristics, NVM is used in the research field as high-speed storage device [23, 24], persistent cache [25], or main memory extension for persistent data structures [22, 26]. However, it is necessary to support transactional updates for consistency guarantee while managing persistent data in NVM. However, there exist several challenges in supporting transactional updates. First, due to the CPU cache, changes on NVM data may not instantly updated. This problem results in performance overhead from instructions like *clflush* for every NVM update. Secondly, NVM failure atomic write size is restricted to 8 bytes. Thus, it cannot guarantee atomicity if an update larger than 8 bytes occurs. This problem can be resolved by implementing commit mechanisms [23, 24, 26], or applying Intel TSX Extension [27], but it incurs additional performance overhead. Lastly, there are write ordering problems for sequential updates. It happens due to CPU's memory write instruction reordering. This can be solved by using memory fence instruction such as *mfence*, which constrains memory update reordering [22, 26].

3 DESIGN AND IMPLEMENTATION

3.1 Design Goals

The design goals of this study for parallel I/O write support in F2FS are:

- **Support parallel writes to a single file.** In a data intensive application such as DBMS, parallel reads for a single file are effectively handled by allowing concurrent accesses to the page cache. However, parallel writes for a single file are not concurrently processed. Linux VFS layer can process parallel writes for a single file simultaneously, but, parallel write requests to the same file are serialized by the file system. This is because a file system such as F2FS, Ext4, XFS, ZFS, and Btrfs can only process one write request at a time for a single file using `inode mutex` [28]. Thus, it should be able to perform multiple writes to a single file in order to effectively process parallel I/Os for a single file. We achieve this goal by adopting *Range Lock* instead of `inode mutex` when it processes the write request to a file.
- **Reduce *Fsync* latency.** The *Fsync* system call is used to persist data that has been written so far. In particular, database applications periodically call *Fsync* to provide application level consistency. For example, database applications can ensure that transactions are stored safely in a storage device only with *Fsync* . This prevents data loss that may occur in case of a power failure. In Manycore server environments, as the number of cores grows, more I/O transactions with *Fsync* calls occur more frequently. In such an environment where *Fsync* is more frequently called, the latency of *Fsync* determines the transaction processing performance of the entire application. Therefore, it is necessary to minimize the latency of *Fsync* while processing the above-mentioned parallel writes. We achieve this goal by adopting small non-volatile memory (NVM) that provides byte-addressability, persistency, and low access memory latency as an extended storage space to disk to save file metadata and file system metadata.

- **Remove I/O blocking on periodic checkpointing.** In F2FS, checkpointing is a process for flushing modified file data, file metadata, and file system metadata from memory to disk to recover from system or power failures. When *Fsync* is called, it checks if 60 seconds have elapsed since the last checkpoint, and if 60 seconds have passed, it triggers a checkpointing. However, during checkpointing, all normal I/O write operations that try to change node logs of the file system including file creation are blocked. Thus, I/O throughput of applications can dramatically drop during the checkpointing process. In addition, checkpointing flushes all dirty data in memory other than the file called by *Fsync*. This increases the I/O checkpointing blocking time, further degrading the application I/O throughput. Therefore, I/O delay time that occurs when checkpointing is called should be minimized. We mitigate this I/O blocking problem when using checkpointing by moving file system metadata and file metadata to NVM.

To address the aforementioned issues, we propose file-level *Range Lock* and *NVM Node Logging*, which are explained in detail in the following sections.

3.2 File-level Range Lock

POSIX-based native file systems such as Ext4, XFS, and F2FS serialize parallel writes to the same file. Currently, the native file systems process only one write at a time using the `inode mutex` in the write operation even though each thread writes to a different area of the file. However, this incurs a huge overhead when processing a large number of writes to the same file.

Simply removing the `inode mutex` lock from the write operation path in the file system is not desirable because it may lead to data inconsistency as multiple threads will modify data in the overlapping ranges simultaneously. On the other hand, the POSIX write ordering constraint will be violated. POSIX requirements on write I/Os are written as the followings in [29].

> *After a write() to a regular file has successfully returned,*
>
> (1) *Any successful read() from each byte position in the file that was modified by that write shall return the data specified by the write() for that position until such byte positions are again modified.*
> (2) *Any subsequent successful write() to the same byte position in the file shall overwrite that file data.*

In order to meet these requirements, we propose to use a file-level *Range Lock* in F2FS that allows running parallel writes to a single file rather than an `inode mutex` lock. The file-level *Range Lock* is able to atomically acquire a lock for only the range of the file to be written. It does not block writes to areas other than the blocks currently being written,

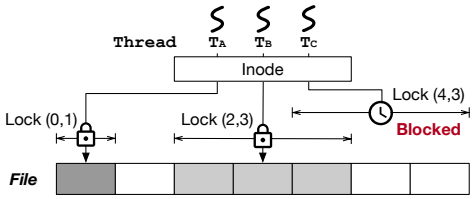


Figure 2: File access using *Range Lock*. Each box of the file represents a data block.

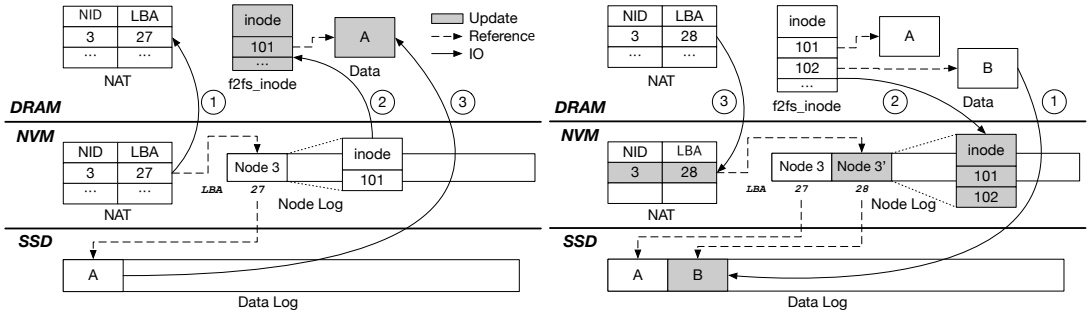
which eventually increases write parallelism when multiple writes are performed on the same file. A write is blocked only when a new write is generated in the range currently being written, thus data consistency can be maintained.

Figure 2 shows an example of how parallel writes are performed using *Range Lock*. Suppose there are three threads, named $Thread_A$ (T_A), $Thread_B$ (T_B), and $Thread_C$ (T_C). Each thread runs with the following order: T_A first writes a file, then, T_B, and then T_C wants to write the file. And T_B does not end before T_C tries to write the file. In this situation, the operating mechanisms of thread A, B, and C are as follows.

(1) T_A starts write operation on the first block of the file. Because there are no other threads which perform write I/O on the first block, T_A acquires a lock and performs its task.
(2) T_B starts write operation on the blocks from third to fifth of the file. As there are no overlapping blocks for T_B, it acquires the lock and performs its operation.
(3) T_C starts write operation on the blocks from fifth to seventh of the file. As T_C performs write operation on the overlapping block with T_B, it is blocked until the overlapped blocks are unlocked.

Applying *Range Lock* allows parallelizing write operations while preserving the POSIX ordering constraints. First, POSIX constraint suggests the need for read and write ordering when overlapping between them is present. This requirement must be met in order to perform parallel read-write operations. However, removing `inode mutex` lock for parallel writes does not preserve the POSIX read-write operation sequence constraint. On the other hand, *Range Lock* will only acquire a lock for the particular blocks on which the thread will perform write I/O. In order to increase parallelism, *Range Lock* also supports parallel reads. Read operations for blocks that do not overlap with blocks of write operations can be executed simultaneously unlike the `inode mutex` lock.

Second, POSIX constraint suggests that if two write I/Os perform on overlapping blocks then only one should be allowed to write on the overlapped blocks. Therefore, multiple writes to the overlapping blocks should be serialized. If multiple writes are permitted by simply removing the `inode mutex` lock, write-write serialization may also not be preserved. However, in the *Range Lock*, write I/Os are to be



(a) Read process using fine-grained inode **(b) Write process using fine-grained inode**

Figure 3: Read and write operation process using a fine-grained inode structure.

serialized if there is an overlapping range of blocks in those two write I/Os.

POSIX-based native file systems perform concurrent writes sequentially by using the `inode mutex` lock. Therefore, modification of metadata such as inode is only done by one write operation at a time, and it is consistent. However, if parallel I/O is allowed for a single file, changes to metadata are also made in parallel, which means that the consistency of parallel updates of metadata should be considered. The metadata for F2FS includes NAT, segment information and inode. The NAT manages the mapping between node id and physical location of the node block as shown in Figure 1. F2FS flushes NAT in memory when *Fsync* is called. Therefore, changing the order in which multiple threads write to the same file in the in-memory does not affect NAT's consistency. F2FS manages blocks in segments (consecutive blocks). The segment information uses a bitmap to manage the validity of each of the blocks within the segment. Parallel writes using *Range Lock* allows to have mutually exclusive ranges in the file. Because each thread enters a critical section using the *Range Lock*, more than one thread can not modify the same bit in a valid bitmap at the same time. Therefore, the consistency of segment information due to parallel writing to a single file is not a problem. Inode consists of information related to a file pointers to the block addresses of the data. When updating the file metadata, the file system uses the `inode mutex` lock and maintains inode consistency. When updating pointers to data in the file, it has to acquire a *Range Lock*, thus it can also keep the inode consistency. *Range Lock* can be applied to file systems that have inode locking problems that occur when multiple threads write to a single file.

3.3 NVM Node Logging

Fsync system call ensures data consistency in case of failures and applications periodically trigger *Fsync* to protect data. Even if parallel writes can increase an overall I/O performance, *Fsync* can greatly degrade the I/O performance. Thus, reducing *Fsync* latency is critical. *NVM Node Logging* mitigates high latency issues in *Fsync* and attempts to reduce I/O blocking time during checkpointing in a Manycore server environment. This can be accomplished by placing file metadata log (Node log) and file system metadata such as NAT on small NVM. In the *NVM Node Logging* approach, only Node log and file system metadata are stored in NVM, and data log is stored in the SSD because our goal of this study is to use small NVM to minimize the performance overhead by *Fsync* and checkpointing operations. With NVM, *Fsync* latency can decrease as it is a write operation on NVM rather than on the SSD block device. Besides, NAT is stored in NVM, thus reducing performance degradation during checkpointing.

Figure 3 shows the read and write procedure using Node log on NVM. The read operation is as follows. First, read a Node log entry via NAT on NVM. Second, identify an LBA corresponding to data in the Node log entry. Lastly, read data block from SSD using LBA. In the case of data write, the data log is firstly stored on an SSD to ensure file system consistency. When *Fsync* is called, the Node is flushed to the NVM's node log. Then, when the checkpoint starts, all Nodes that are not yet flushed to the NVM are flushed with the NAT. At this moment, just using *memcpy* to store data on NVM does not guarantee consistency. *clflush* is vital to make sure the flush procedure is completed from the cache to memory. Also, to prevent re-ordering while flushing, *mfence* instruction is required.

NVM can be used as a write-back cache [25] or as a journaling device [24, 30]. In the *NVM Node Logging*, NVM is used as storage only for file and file system metadata. Therefore, if NVM capacity is sufficient, existing data or metadata caching approaches can be applied orthogonally with *NVM Node Logging*.

3.4 Fine-grained Inode

F2FS is made for block device where a basic writing unit is 4KB. File metadata, such as inode, direct node and indirect node, is designed to fit a 4KB size. Figure 4(a) shows a traditional inode structure in F2FS. The inode structure consists of inode information from the VFS layer, address

(a) Baseline inode (b) Fine-grained inode

Figure 4: Comparing the inode structure of existing F2FS and fine-grained inode structure.

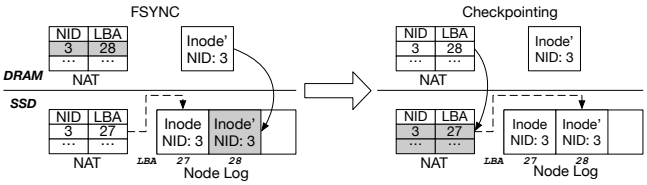
space to indicate data blocks, and NID for node information. If a file is small, the inode only needs direct pointers to the data block. On the other hand, if the file is large, it also uses indirect pointers. However, since the inode is 4KB, it has both direct pointers and indirect pointers regardless of file size. In *NVM Node Logging*, the node is located in NVM. Due to the byte-addressability of NVM, the inode does not need to be sized to 4KB. The size of the inode structure can be small. We call this inode structure a fine-grained inode structure. Figure 4(b) shows the fine-grained inode structure which allocates only required amount of NVM. *Fsync* forces I/O on a block size even though the changed portion is much less than the block size, which can cause a write amplification problem. This write amplification is critical in NVM in terms of NVM's lifetime limitation.

However, adopting *NVM Node Logging* allows access to the NVM in bytes, so that only a portion of the inode structure can be modified. Therefore, the problem of write amplification due to block sized I/Os can be solved. By using a fine-grained inode structure, the size of the inode which needs to be persisted is decreased by 90%, compared to a traditional block size inode structure.

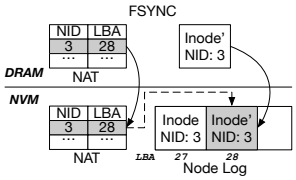
3.5 Pin Point NAT Update

As mentioned earlier, checkpointing is called every 60 seconds. Once checkpointing is called, the entire NAT is flushed. This also causes write amplification problems by flushing unnecessary data to disk. The *Pin Point NAT Update* solves the write amplification problem by flushing only the modified NAT entries to NVM when *Fsync* is invoked.

Figure 5(a) shows how existing F2FS updates and flushes NAT when the NAT is in the SSD. In F2FS, the block address of an entry in NAT is updated when the node log entry is flushed. And when checkpointing is performed, the entire NAT is flushed to the SSD in NAT blocks even though only some of the NAT entries need to be updated. Suppose that the inode of NID 3 is modified in Figure 5(a). When *Fsync* is called, the modified inode will be appended at a new block in the Node log on the SSD. And the corresponding entry (LBA) in the NAT in memory is updated to reflect this new NID-to-LBA mapping entry change. When the checkpointing is called, the entire NAT is flushed. On the other hand,



(a) Baseline approach to flush inode and NAT



(b) Pin Point NAT Update to flush inode and NAT

Figure 5: Procedure for *Fsync* and checkpointing in baseline and procedure for *Fsync* only in Pin Point NAT Update.

Figure 5(b) illustrates the *Pin Point NAT Update* procedure where both the node log entry and modified NAT items are flushed to NVM when *Fsync* is called, thereby minimizing the overhead of flushing the entire NAT to disk and reducing the latency of *Fsync*. The *Pin Point NAT Update* guarantees the atomicity of the flushed data because the size of the NAT entry is smaller than the cache line size.

NVM Node Logging including *Fine-grained Inode* and *Pin Point NAT Update* can also reduce the mapping table flush overhead of file system metadata in other log-based file systems.

4 EVALUATION

4.1 Experimental Setup

For a fair comparison, we configured two different scales of Manycore testbeds. Table 1 and Table 2 show the detail hardware specification of each testbed.

Workloads: Synthetic workload and realistic workload both were used in the evaluation. We use FxMark [28] for Synthetic workload. FxMark is the benchmark that can evaluate I/O performance in Manycore environments, and is able to test performance scalability for various synthetic workload patterns by varying CPU core numbers. FxMark can create many different workload patterns. Especially, DWOM and DWSL workloads were used in our evaluation. DWOM is a workload that multiple threads write 4KB to different offsets of a single file. DWSL is a workload that multiple threads write 4KB to their own files and each thread calls *Fsync* for every 4KB write. We consider both Buffered I/O and Direct I/O configurations for DWOM in order to observe the performance difference between using page cache or not.

For realistic workloads, we use Filebench [31] which is the file system benchmark, and TPC-C [32] with MySQL 5.7 [3]. OLTP and Varmail are used as workloads in Filebench.

Table 1: Configurations for Testbed-I.

CPU	Intel Xeon E5-2640 v4 2.4GHz [33] CPU Node (#): 1 Cores per Node (#): 10
RAM	64GB
SSD	Samsung SSD 850 PRO 256GB (SATA) [34] Read: 550 MB/s, Write: 520 MB/s
NVM	32GB Emulated as PMEM device on RAM
OS	Linux kernel 4.14.11

Table 2: Configurations for Testbed-II.

CPU	Intel Xeon E7-8870 v2 2.3GHz [35] CPU Node (#): 8 Cores per Node (#): 15
RAM	740GB
SSD	Intel SSD 750 Series 400GB (NVMe) [36] Read: 2200 MB/s, Write: 900 MB/s
NVM	32GB Emulated as PMEM device on RAM
OS	Linux kernel 4.14.11

Table 3: OLTP configuration parameters.

	ndbwriters	nshadow	nfiles	runtime
Testbed-I	16	4	1	60sec
Testbed-II	80	40		

Table 4: Varmail configuration parameters.

	nfiles	nthreads	iosize	mean	runtime
Testbed-I	10000	20	1MB	1MB	60sec
Testbed-II		120			

Filebench OLTP mimics I/O patterns in DBMS. In Filebench OLTP, one thread writes log and other threads process I/Os for DB files. Varmail OLTP is the workload that imitates I/O pattern of a simple mail server. Varmail OLTP consists of multiple threads processing create-append-sync, read-append-sync, and read-delete operations. Table 3 and Table 4 show the parameter settings in OLTP and Varmail workloads. TPC-C benchmark uses MySQL, thus it can check the performance of real database application.

We evaluate F2FS write performance with *Range Lock* implementation for parallel write optimization and *NVM Node Logging*. Our evaluation will answer the following questions.

- What are the performance bottlenecks in F2FS with parallel write scenarios?
- How is the performance improved when multiple threads write to a single file compared to baseline F2FS?
- How is the efficiency and performance trend improved with *NVM Node Logging*?
- How optimized F2FS will behave in real workloads?
- When does the disk become a bottleneck, besides file system?

Emulating NVM Device: A part of the contiguous space of the DRAM is set as the NVM area and registered as a device through the PMEM driver. The NVM area is accessed via direct access (DAX) and memcpy is used to read/write at the virtual address of NVM.

For evaluations, we compare different implementations as below.

- F2FS(Baseline): F2FS included in Linux kernel.
- F2FS(RL): F2FS with *Range Lock* mechanism.
- F2FS(NL): F2FS with *NVM Node Logging* mechanism.
- F2FS(Integrated): F2FS with both *Range Lock* and *NVM Node Logging* mechanisms.

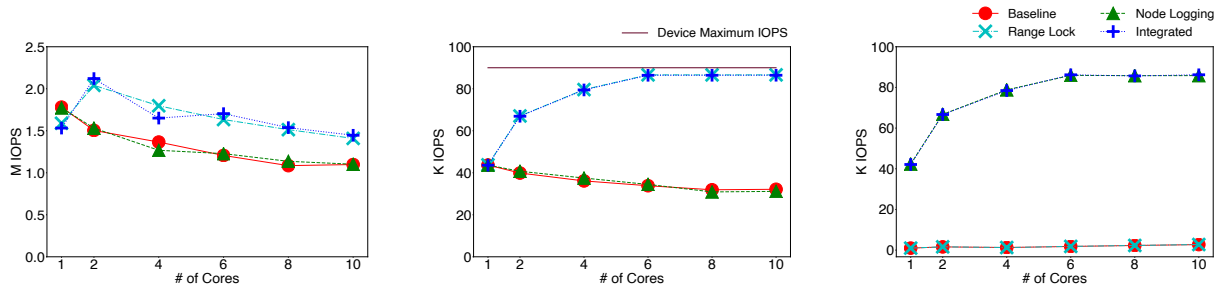
4.2 Evaluating Scalability for Various Workload Patterns

4.2.1 Experiment with a single CPU node based Manycore server. To verify our proposed design, we measure scalability using DWOM and DWSL workloads of FxMark with Testbed-I. Figure 6(a) shows the throughput of DWOM workload with Buffered I/O varying number of cores. Throughput of F2FS(RL) and F2FS(Integrated) increase 41% compared to Baseline in all cases after 2 cores. Because of *Range Lock*, multiple threads can write into a single file simultaneously in F2FS(RL) and F2FS(Integrated). F2FS(NL) performs similarly with Baseline because it uses `inode mutex` as Baseline does. There is no benefit to using *NVM Node Logging* because there is no *Fsync* call in DWOM. When only one core is enabled, both F2FS(RL) and F2FS(Integrated) show lower throughput than Baseline. Also because of the higher management overhead of *Range Lock* compared to `inode mutex`, throughput decreases when there is no parallel writes. Due to the management cost of *Range Lock*, throughput does not scale as number of cores increase.

Figure 6(b) shows the throughput of DWOM workload with Direct I/O varying number of cores. Throughput of F2FS(RL) and F2FS(Integrated) increases 2.7 times compared to Baseline. Because Direct I/O enforces to access to SSD device which is slower than page cache access, performance improvement by *Range Lock* is clearly shown and *Range Lock* management overhead is hidden by higher device access latency. Because the SSD device' is saturated, throughput does not scale after 6 cores.

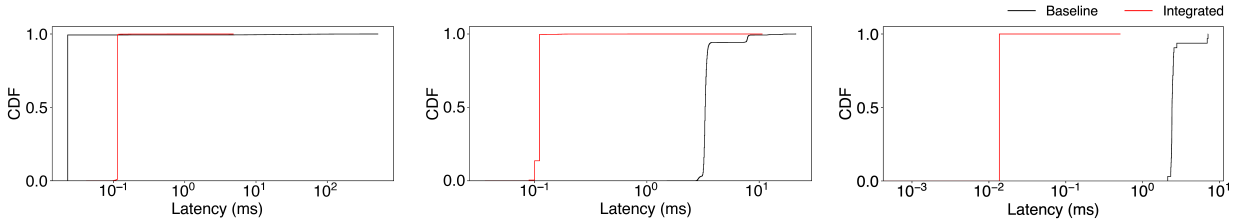
Figure 6(c) shows throughput of DWSL workload varying number of cores. In F2FS(NL) and F2FS(Integrated), Node log I/Os occurred by *Fsync* are performed on NVM, which reduces *Fsync* latency while increasing throughput. F2FS(NL) and F2FS(Integrated) show 39.6 times performance improvement compared to Baseline. Because in DWSL workload, multiple threads write their own private files, *Range Lock* can not improve throughput. As shown in DWOM, throughput does not scale after 6 cores due to device bandwidth saturation.

4.2.2 Analysis Latency of Range Lock & NVM Node Logging. Figure 7(a) shows CDF of write latency in DWOM for



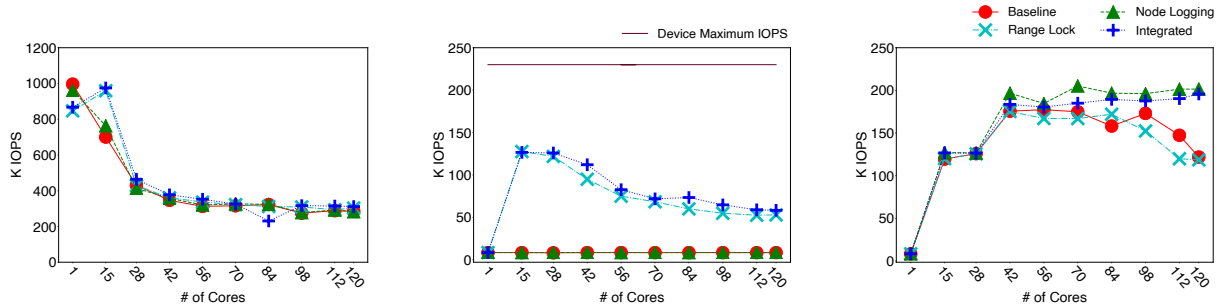
(a) DWOM (Buffered I/O) (b) DWOM (Direct I/O) (c) DWSL (Buffered I/O)

Figure 6: Performance comparison of various synthetic workloads (DWOM, DWSL) with Testbed-I.



(a) CDF for Write latency on DWOM (b) CDF for *Fsync* latency on DWSL (c) CDF for Checkpoint latency on DWSL

Figure 7: Write, *Fsync*, and Checkpoint latency comparison of F2FS-RN with Testbed-I.



(a) DWOM (Buffered I/O) (b) DWOM (Direct I/O) (c) DWSL (Buffered I/O)

Figure 8: Performance comparison of various synthetic workloads (DWOM, DWSL) with Testbed-II.

Baseline and F2FS(Integrated). In case of Baseline, tail latency is quite long due to `inode mutex` bottleneck. While F2FS(Integrated) shows lower tail latency since *Range Lock* allows multiple threads to write on a single file. Figure 7(b) shows CDF of *Fsync* latency in DWSL for Baseline and F2FS(Integrated). *NVM Node Logging* mechanism performs metadata I/O from *Fsync* quite rapidly, and shows reduced tail latency. Figure 7(c) shows checkpoint latency of Baseline and F2FS(Integrated) in DWSL. F2FS(Integrated) reduces I/O in checkpointing by Pin Point NAT Update, and reduced tail latency can be observed.

4.2.3 Experiment with a NUMA node-based Manycore server. Figure 8(a)(b) shows throughput of FxMark DWOM with respect to increasing number of cores in Testbed-II. Figure 8(a) shows experiment results for DWOM that uses Buffered I/O. Its result coincided with the results of Testbed-I. With 7 cores, throughput of F2FS(RL) and F2FS(Integrated) improved 44% compared to Baseline, but did not show improvement in most cases. Figure 8(b) shows experiment result of DWOM that uses Direct I/O. Throughput of F2FS(RL) and F2FS(Integrated) improved up to 14.9% compared to Baseline. However, maximum throughput was shown in 15 cores, and when more cores were used, throughput declined instead. This graph only shows IOPS value of data. However, 4KB of extra operation occurs since change in inode is followed by data modification. Hence, including this, it can be shown that IOPS reaches SSD device max bandwidth when 15 cores are used. However, as number of cores increases, more I/O needs to be taken care of while device cannot perform higher than maximum bandwidth. It can be presumed that performance decreases due to SSD's internal queuing delay. Figure 8(c) shows throughput of DWSL in FxMark,

Table 5: OLTP results on Testbed-I.

OLTP		Baseline	Range Lock	Node Logging	Integrated
Total ops/s		20082	33004	102084	102209
Write ops/s		11985	19692	60891	60966
Read ops/s		7975	13113	40572	40622
dbwrite	Avg	0.003	0.003	0.002	0.001
Latency	Max	39.9	39.9	35.7	39.1

Table 6: Varmail results on Testbed-I.

Varmail		Baseline	Range Lock	Node Logging	Integrated
Total ops/s		4703.9	4647.0	5493.1	5506.2
fsync ops/s		362	357.5	422.5	423.5
fsync	Avg	26.5	26.6	22.3	22.3
Latency	Max	118.6	154.2	115.2	96.4

Table 7: OLTP results on Testbed-II.

OLTP		Baseline	Range Lock	Node Logging	Integrated
Total ops/s		328546	313570	327300	331497
Write ops/s		22950	23875	24543	25057
Read ops/s		305275	289370	302421	306098
dbwrite	Avg	0.01	0.01	0.01	0.01
Latency	Max	187.5	111.7	199.8	55.8

Table 8: Varmail results on Testbed-II.

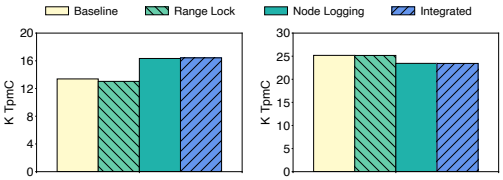
Varmail		Baseline	Range Lock	Node Logging	Integrated
Total ops/s		8056.2	7898.6	7780.9	8007.3
fsync ops/s		619.5	607.5	598.5	615.5
fsync	Avg	91.4	93.7	95.2	91.9
Latency	Max	337.6	338.3	311.9	315.5

with respect to number of cores. Like Testbed-I, it can be confirmed that throughput increases from *Fsync* latency improvement through *NVM Node Logging* in F2FS(NL) and F2FS(Integrated). F2FS(NL) and F2FS(Integrated) improved up to 35.8% compared to Baseline. However, throughput of F2FS(Integrated) is lower than that of F2FS(NL) due to the overhead of *Range Lock* management.

4.3 Evaluating Scalability for Realistic Workloads

Table 5 provides experimental results for OLTP workload in Filebench with Testbed-I. The DB write throughput increased by approximately 64% with F2FS(RL) compared to Baseline. This is because, *Range Lock* allows multiple threads to access a single DB file simultaneously. As the write throughput of DB files increased, the read throughput of DB files also increased proportionally. F2FS(NL) throughput increased up to 5× compared to the Baseline. OLTP workloads use *dsync* mode to perform writes. F2FS calls *Fsync* on every write. Significant performance improvements in F2FS (NL) are due to a large number of these *Fsync* calls in the workload. In F2FS(Integrated), *Range Lock* made a slight increase in throughput. But, overall throughput is almost the same as F2FS(NL) because the throughput increase by *NVM Node Logging* is dominant compared to *Range Lock*.

Table 6 shows the experimental results for Varmail in Filebench in Testbed-I. Compared to Baseline,



(a) Testbed-I **(b) Testbed-II**

Figure 9: Performance comparison of various F2FS optimizations for TPC-C.

F2FS(Integrated) showed about 16% increase in throughput. In Table 6, the average latency of F2FS(NL) *Fsync* decreased by 19% and the maximum latency decreased by up to 10%. Therefore, the latency reduction of *Fsync* due to *NVM Node Logging* was confirmed. On the other hand, since there is no task that multiple threads writing a file simultaneously, we can not see performance improvement through *Range Lock*.

Table 7 shows the experiment results for Filebench OLTP in Testbed-II. Unlike Testbed-I, overall throughput improvement was not seen when compared to Baseline. Looking at the throughput of the DB-write alone, F2FS(RL) showed a performance improvement of approximately 4% over the Baseline through *Range Lock*, but there was no increase in overall throughput due to reads which do not show improvement taking up the majority of total I/Os. Also, the maximum latency of DB-write in F2FS(RL) is reduced by 41%. F2FS(NL) also showed no difference in overall throughput compared to Baseline, but DB-write throughput increased by about 6%. No improvement in maximum latency was observed in F2FS(NL). F2FS(Integrated) also showed similar throughput to Baseline. DB-write showed 9% throughput increase and tail latency decrease by 71% compared to Baseline. *Range Lock* has solved the problem of write serialization, showing the improvement of DB-write performance and reducing the tail latency.

Table 8 shows the experimental results for Varmail in Filebench in Testbed-II. Unlike the results in Testbed-I, F2FS(RL), F2FS(NL) and F2FS(Integrated) showed similar throughput with the Baseline. Rather, the throughput is slightly lowered in F2FS(NL). Because Varmail has operations other than write and *Fsync*, such as file creation, read, and removal, we can not see throughput improvement through the proposed methods in Testbed-II, which uses SSDs fast enough. In addition, the minimum latency and maximum latency of F2FS(NL) and F2FS(Integrated) using *NVM Node Logging* are slightly lowered. However, there is no significant difference in overall latency and the latency of the F2FS(NL) is slightly higher resulting in lower throughput.

Figure 9(a) shows the TpmC change for TPC-C in Testbed-I. At F2FS(NL) and F2FS(Integrated), the throughput improved by approximately 22% compared to the Baseline. On the other hand, F2FS(RL) shows similar throughput as Baseline.

Because TPC-C is a benchmark using actual MySQL, it processes queries through the database and stores the results in a log file. Data stored in the logs is stored in a DB file through background I/O. Testbed-I is not a highly compute-intensive environment because it has only 10 cores. Therefore, compute for query processing at the DBMS and I/O for storing logs are prominent and background I/O is not noticeable. The performance improvements in these situations cannot be expected because *Range Lock* is the optimization of shared writes that occur when log is written to a DB file.

Figure 9(b) shows the results of the experiment on TPC-C in Testbed-II. F2FS(RL) has not resulted in throughput improvements for TPC-C workloads, such as in Testbed-I. Unlike Testbed-I, the performance of F2FS(NL) and F2FS(Integrated) slightly decreases. TPC-C is a highly SELECT-intensive workload with read I/O, and it is suspected that the DBMS is producing I/O that can be adequately handled by high performance storage device. As a result, we couldn't see any improvement in throughput through the techniques proposed in this paper.

Through the results of micro and macro benchmark in Testbed-II, *Range Lock* and *NVM Node Logging* techniques proposed in this paper were found to be suitable for a highly write intensive workload. Like micro benchmark, which issues a lot of write operations, performance improvements were seen well. However, in hardware resource-rich systems with write-less workload, performance improvements could not be seen. Moreover, the performance decreased slightly.

5 RELATED WORK

Lock Optimizations: I/O performance optimizations of the application using parallel I/O have been exploited in HPC and distributed computing systems [8, 9, 37, 38]. Particularly, range lock has been introduced to minimize delays and waiting time in parallel I/O scenarios [39, 40]. Thakur et al. [39] proposed a Byte-Range Locking scheme on a file system layer to optimize multiple write operations while ensuring atomicity semantics for non-contiguous file accesses in high-performance MPI-IO ROMIO. Bennett et al. [40] proposed another Byte-Range Locking scheme that changes the level according to the complexity of locking in a client-server environment. Parallel distributed file systems such as Lustre [41] and Gluster [42] are equipped with range lock to speed-up parallel I/Os while maintaining the consistency semantics. Both studies parallelize the I/Os of several threads simultaneously accessing a file by using the range lock technique and preserve file consistency. However, most POSIX-based native file systems use `inode mutex` when writing files, so they cannot fully attain parallelism in parallel I/O cases. In this paper, we extend the range locking method used in HPC and distributed computing systems to the native file system to improve write performance of the parallel I/O.

Fsync Optimizations: Several existing approaches such as optimization at journaling and on storage backends tried to overcome *Fsync* performance degradation [30, 43, 44]. iJournaling [43] proposed transaction processing on a file-by-file basis to solve the performance degradation caused by compound transaction processing that occurs when *Fsync* is called from the journaling file system. However, iJournaling still has high overhead with increasing number of transactions to process when *Fsync* is called on multiple files. RFLUSH [44] proposed a new instruction to improve the lump-sum problem of storing all the data in the buffer inside the storage device when the Flush command is executed. The RFLUSH instruction is a fine-grained FLUSH instruction that stores only the data for the file called *Fsync* in the internal buffer of the storage device. So, it can reduce unnecessary data storage and utilize caching due to buffering for other data. Existing studies are either journaling level or device level *Fsync* optimizations. In this paper, we propose the file system level optimization, which makes additional I/O caused by *Fsync* to be fast with NVM. Our work is orthogonal with device level *Fsync* optimization like RFLUSH.

Write Optimizations: There exist several studies to improve I/O performance of the file system using NVM [24, 25, 30]. UBJ [30] integrates the page cache and journal area on NVM. It reduces the memory copy and space overhead for journaling. Tinca [25] proposed a technique to use NVM as a block cache layer. Fine-grained Metadata Journaling [24] proposed a technique to use non-volatile memory as journal area. It solves the write amplification problem by writing the journal to the necessary part in bytes utilizing the byte-addressability of non-volatile memory. ScaleFS [45] proposed a per-core operation log to address low file system scalability due to cache-line conflict problems when updating in-memory copies of directory blocks. The cache-line conflict problem was solved by logging the directory changes to the per-core log and merging them in a delayed way using global timestamps.

6 CONCLUSION

We identified the cause of the performance bottleneck in F2FS for parallel I/O workloads on Manycore servers, which are manifolded: First, in F2FS, when multiple threads write on a file, F2FS serializes them so even if each thread writes on different region of the file, one has to wait until another finishes. Second, when many threads call writes followed by *Fsync*, file metadata (inode) and file system metadata (NAT) flush operations can burden file system performance. In order to solve these problems, we propose to implement *Range Lock* and *NVM Node Logging* using small NVM in F2FS. With extensive evaluations on the Manycore servers, we witnessed significant improvements in F2FS's write performance for write-intensive and high shared file write workloads.

ACKNOWLEDGMENTS

We thank the reviewers and our shepherd, Anirudh Badam for their constructive comments that have significantly improved the paper. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System).

REFERENCES

[1] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A Scalable Storage Manager for the Multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*, 2009, pp. 24–35.

[2] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[3] MySQL. [Online]. Available: https://www.mysql.com/

[4] SQLite. [Online]. Available: https://www.sqlite.org/

[5] RocksDB. [Online]. Available: https://rocksdb.org/

[6] LevelDB. [Online]. Available: http://leveldb.org/

[7] Bouteiller, Lemarinier, Krawezik, and Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *Proceedings of the 2003 IEEE International Conference on Cluster Computing (CLUSTER)*, 2003, pp. 242–250.

[8] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2003, pp. 39–48.

[9] Y. Yu, D. H. Rudd, Z. Lan, N. Y. Gnedin, A. Kravtsov, and J. Wu, "Improving Parallel IO Performance of Cell-based AMR Cosmology Applications," in *Proceedings of the 26th IEEE International Conference on Parallel & Distributed Processing Symposium (IPDPS)*, 2012, pp. 933–944.

[10] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: Multithreaded MPI for Many-Core Environments," in *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*, 2014, pp. 125–134.

[12] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[13] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time Traveling Optimistic Concurrency Control," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016, pp. 1629–1642.

[14] H. Kimura, "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 691–706.

[15] Performance Benchmarking for PCIe and NVMe Enterprise Solid State Drive. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-pcie-nvme-enterprise-ssds-white-paper.pdf

[16] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proceedings of the 2007 Linux Symposium*, 2007, pp. 21–33.

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

[18] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 273–286.

[19] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, p. 22, 2017.

[20] M. J. Breitwisch, "Phase Change Memory," in *Proceedings of the 2008 International Interconnect Technology Conference (ITTC)*, 2008, pp. 219–221.

[21] T. Kawahara, "Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 52–63, 2011.

[22] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi, "Log-Free Concurrent Data Structures," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2018, pp. 373–386.

[23] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 323–338.

[24] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proceedings of the 32nd IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2016, pp. 1–13.

[25] Q. Wei, C. Wang, C. Chen, Y. Yang, J. Yang, and M. Xue, "Transactional NVM Cache with High Performance and Crash Consistency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 56:1–56:12.

[26] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017, pp. 257–270.

[27] Intel Transactional Synchronization Extensions (Intel TSX) Overview. [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview

[28] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding Manycore Scalability of File Systems," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2016, pp. 71–85.

[29] The Open Group Technical Standard Base Specifications, Issue 7 - POSIX.1-2017. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[30] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013, pp. 73–80.

[31] Filebench Benchmark. [Online]. Available: https://github.com/filebench/filebench

[32] TPC-C Benchmark. [Online]. Available: https://github.com/Percona-Lab/tpcc-mysql

[33] Intel Xeon Processor E5-2640 v4. [Online]. Available: https://ark.intel.com/products/92984

[34] Samsung 850 PRO Series SSD. [Online]. Available: https://www.samsung.com/us/business/support/owners/product/850-pro-series-256gb/

[35] Intel Xeon Processor E7-8870 v2. [Online]. Available: https://ark.intel.com/products/75255/Intel-Xeon-Processor-E7-8870-v2-30M-Cache-2-30-GHz-

[36] Intel SSD 750 SERIES. [Online]. Available: https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series/750-400gb-aic-20nm.html

[37] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009, pp. 21:1–21:12.

[38] Infinite Memory Engine (IME). [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache/

[39] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the 6th ACM Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, 1999, pp. 23–32.

[40] R. B. Bennett, B. P. Dixon, and E. Johnson, "Byte Range Locking in A Distributed Environment," Sep. 21 1999, US Patent 5,956,712.

[41] P. Schwan *et al.*, "Lustre: Building a File System for 1000-node Clusters," in *Proceedings of the 2003 Linux symposium*, 2003, pp. 380–386.

[42] The Gluster File System. [Online]. Available: http://www.gluster.org/

[43] D. Park and D. Shin, "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017, pp. 787–798.

[44] J. Yeon, M. Jeong, S. Lee, and E. Lee, "RFLUSH: Rethink the Flush," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018, pp. 201–210.

[45] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scaling a file system to many cores using an operation log," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 69–86.