Concurrent File Metadata Structure Using Readers-Writer Lock

Chang-Gyu Lee
Sogang University
Seoul, Republic of Korea
changgyu@sogang.ac.kr

Sunghyun Noh*
Sogang University
Seoul, Republic of Korea
nsh0249@sogang.ac.kr

Hyeongu Kang
Sogang University
Seoul, Republic of Korea
hyeongu@sogang.ac.kr

Soon Hwang
Sogang University
Seoul, Republic of Korea
hs950826@sogang.ac.kr

Youngjae Kim†
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

ABSTRACT

Linux file systems serialize threads when writing shared files. Recent studies have attempted to adopt range locks on shared files to solve this serialization problem, allowing file I/O to be executed concurrently. However, we have found that even with a range lock, I/O throughput no longer increases after a certain number of cores and decreases rapidly on a manycore server. Through extensive performance profiling, we found the *cascading tree lock problem* that serializes concurrent accesses to the file metadata structure. A mutex lock-based locking mechanism for each file metadata structure serializes I/O requests in modern Linux file systems such as F2FS. In this paper, we present *nCache*, a novel file metadata cache framework using readers-writer lock that allows concurrent I/O operations for the shared file. *nCache* solves the I/O scalability problem in the manycore server while ensuring consistent updates. We implemented *nCache* in F2FS and evaluated it using FxMark on a 120-core server with high-performance NVMe SSDs. Our extensive evaluations show that *nCache* achieves maximum device throughput in FxMark's shared file I/O workload. It also shows 4.1x higher throughput compared to the baseline F2FS with range locks for realistic workloads.

CCS CONCEPTS

• **Software and its engineering → File systems management**; *Massively parallel systems*; • **Information systems → Directory structures**;

KEYWORDS

Operating System, File System, Concurrency

*S. Noh is currently affiliated with Samsung Electronics. This work was conducted when he was with Sogang University.
†Y. Kim is the corresponding author.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3441992

ACM Reference Format:
Chang-Gyu Lee, Sunghyun Noh, Hyeongu Kang, Soon Hwang, and Youngjae Kim. 2021. Concurrent File Metadata Structure Using Readers-Writer Lock. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3441992

1 INTRODUCTION

Manycore servers are equipped with hundreds of CPU cores. The Intel Optane DC memory manycore server has 56 cores with two CPU sockets on a single server [28], and the IBM manycore server has 120 cores with eight CPU sockets [12, 20]. With a large number of cores in a single server, a highly parallel application generates massive parallel I/O to exploit higher storage throughput. For instance, databases [10, 13, 29, 30], scientific simulations [3, 11, 17, 25, 31], and machine learning frameworks aggressively utilize massive parallelism while generating a vast amount of parallel I/O for user data, training data, and simulation checkpoints. Parallel I/O from these applications usually falls into private file I/O or shared file I/O. In private file I/O, each thread performs I/O on its own private file. Each thread has exclusive access to that file, so there is little contention between I/O threads in the file system. Thus, the I/O throughput of the file system increases as the number of I/O threads increases. On the other hand, in shared file I/O, multiple threads share a single file, competing for the file metadata. Threads need synchronization when accessing the shared file, so I/O throughput does not increase with the number of I/O threads due to the file sharing contention between threads in the file system.

When multiple threads write shared files, Linux file systems such as Ext4 and F2FS [15] serialize thread's file accesses. To solve this problem, the file system adopts a range lock, granting access to the file if a thread accesses a non-overlapping range of the file [12, 16]. However, we observed that there is a limit to increasing I/O throughput. To assess the scalability of the range lock in the file system, we evaluated F2FS with two types of range lock implementations (interval tree-based range lock [16] and atomic operation-based range lock [12]) on the IBM 120-core manycore machine using the shared file write workload in FxMark [20]. Detailed experimental setups are described in Section 5.1. Figure 1 shows the experiment results of these two range lock implementations on F2FS. Both types of range lock implementations ($F2FS_{RL}$ and $F2FS_{AT}$) saturate the maximum device bandwidth at 15 cores. However, the throughput drops rapidly after 15 cores. Accordingly, the range lock fails

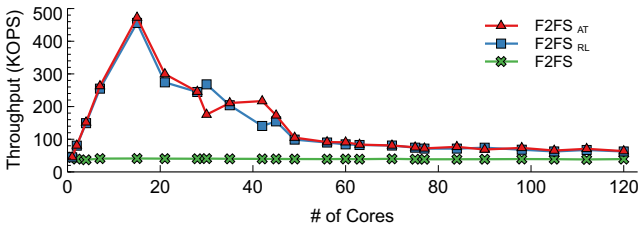


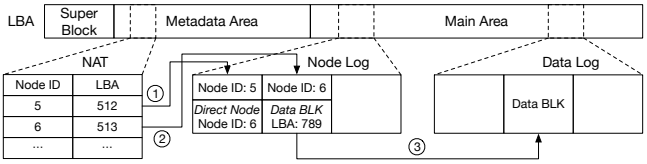
Figure 1: Comparison of default F2FS and two range lock implementations for F2FS for shared file I/O. In the legend, AT and RL denote interval tree-based range lock and atomic operation-based range lock, respectively.

to scale. The same trend in both types of range lock implementations implies that the scalability bottleneck is not in the range lock primitive itself.

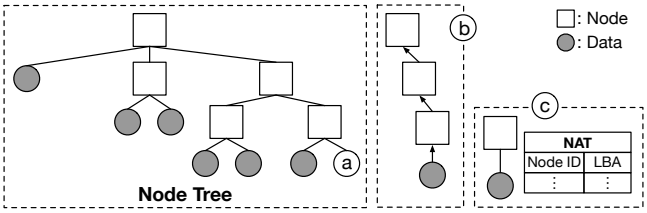
We further analyzed the CPU cycle consumption of file system functions using the same workload. Most of the CPU cycles are consumed for searching the corresponding block address from the file metadata and waiting for lock acquisition for it. 14.54% of the total CPU cycles are consumed for accessing file metadata at 15 cores, while the device maximum throughput is saturated. On the other hand, 51.37% of total CPU cycles are used for accessing file metadata at 120 cores. From this result, we identified that the performance drop is due to the lack of concurrency in the file metadata structure.

Writing a file in a file system consists of (i) accessing file metadata and (ii) writing file data. The range lock enables concurrent file data I/O on the shared file but does not help parallelize file metadata accesses. That is, accessing file metadata is still serialized. We reveal that the serialization problem in the file metadata access is the main cause of the bottleneck for I/O scalability in file system using the range lock for shared file I/O workloads. Since the file metadata structure is fundamentally a tree, most file systems use the mutex lock on file metadata structures to prevent inconsistent updates. To solve this, we employ readers-writer lock (RW lock) to achieve concurrent file metadata access. Thus, in this paper, we propose *nCache*, a novel file metadata cache framework for concurrent file metadata access using RW lock. Our work makes the following specific contributions:

- **Cascading Tree Lock Problem:** We identified a cascading tree lock problem of file metadata. In Linux file systems, when multiple threads access the same file metadata, every node of the file metadata is protected using the inode mutex lock, and they are all serialized. This problem is called the cascading tree lock problem. In addition, we described the bottleneck that occurs when accessing file metadata in F2FS as an example of updating the *Node Tree* in F2FS.
- ***nCache* Framework:** We aimed to design a concurrent file metadata structure where multiple threads access the file metadata concurrently and update it consistently. For this, we proposed *nCache*, which employs readers-writer lock (RW lock) on the file metadata (every node in *Node Tree*). *nCache* allows concurrent access to the shared file by threads. We have also identified possible inconsistent update scenarios when updating file metadata. For consistent file metadata updates, *nCache* adopts double-checked locking [19, 22, 23].
- **Implementation and Evaluation:** For evaluation, we implemented *nCache* for various versions of F2FS with a range lock in Linux kernel version 4.1.4. As a representative result, F2FS, which implements atomic range lock and *nCache* (denoted as $F2FS_{AT+NC}$), saturated all available write and read bandwidths of the Intel Optane SSD 900P device in DWOM and DRBM workloads, which are shared file I/O workloads (write and read workloads respectively) of the FxMark benchmark [20]. We also ran experiments using HACC-IO [26], which is an I/O benchmark for HACC [6] and a representative parallel shared file I/O workload in HPC. The results show that $F2FS_{AT+NC}$ offers scalable throughput as the number of cores increases on the manycore server.



(a) F2FS's on-disk data structure.



(b) inode structure in the page cache.

Figure 2: On-disk data structure of the F2FS file system and inode structure in the page cache.

2 BACKGROUND

In this section, we describe the on-disk data structure of F2FS, the inode structure of the page cache (*Node Tree*), Node Address Translation (NAT) of F2FS, and range lock.

2.1 F2FS File System

F2FS [15] is a log-structured file system optimized for a solid-state drive (SSD). F2FS has two types of logs which are Node Log and Data Log. In F2FS, *file metadata* such as inode, direct node, and indirect node are called *Node*. The inode maintains file-specific metadata such as the latest access time and a set of logical block addresses. When the inode cannot hold the entire set of block addresses in a fixed-size block because the file is big, the inode utilizes direct node and indirect node. The indirect node stores block addresses of direct nodes, and the direct node stores block addresses of *file data*. The inode can address huge files by storing direct and indirect nodes and searching the block addresses of the *file data* following a chain of indirect and direct nodes.

Nodes are appended at the end of the Node Log when they are updated. Nodes are distinguished by a unique identifier called a

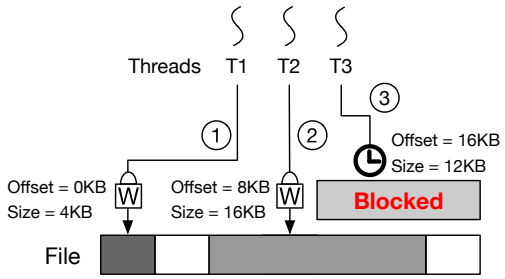


Figure 3: Example of range lock use with writer threads.

Node ID. The *file data* is appended to the Data Log and referenced by Nodes. F2FS uses Node ID to read a Node in the Node Log. That is, Node ID has to be translated into a block address to access a Node. To this end, F2FS maintains a mapping table called a Node Address Table (NAT) in the file system-wide metadata. Figure 2a shows on-disk structures and Node ID translation via NAT. For example, consider the case of reading a single data block in the middle of a file. First of all, F2FS searches the block address of the inode using its Node ID in NAT (①). Because the target data block is under the direct node, not the inode, F2FS takes the Node ID of the direct node. Then the Node ID of the direct node is translated by NAT into the block address again (②). Finally, F2FS retrieves the block address of the data block from the direct node and then reads the data (③).

2.2 *Node Tree* and NAT

The key difference between F2FS and previous log-structured file systems comes from the NAT. The indirection using Node ID reduces recursive updates of Node in the log-structure file system. Figure 2b shows the inode structure of a file in the page cache. It is represented as a tree. For the convenience of explanation, we call this tree *Node Tree*. Every Node in *Node Tree* represents a Node or Data in F2FS. In other words, the inode is denoted by the root of *Node Tree*. Direct or indirect nodes are denoted by internal nodes of the tree, and data blocks of the file are denoted by leaf nodes. Let's consider when a portion of a file is updated (ⓐ). Since F2FS is a log-structured file system, all updates will be out-of-place updates. A new leaf node is created and then added to *Node Tree*, and ancestor nodes are accordingly required to be updated in an out-of-place manner. If its block address points nodes as traditional log-structured file systems, all nodes in the path from the root to the leaf node will be updated (ⓑ). However, in F2FS, only the parent node and its block address in NAT will be updated due to Node ID indirection (ⓒ).

2.3 Range Lock

Parallel I/O is one of the core techniques to maximize I/O throughput. The increase in the number of CPU cores and the parallelism of storage media has led to single servers generating massively parallel I/O workloads. However, Linux file systems such as EXT4 and F2FS showed poor performance in parallel I/O on a shared file [20]. This is because the current implementation of the file system uses mutex lock on inode for data consistency and accordingly serializes I/Os to a shared file.

To enable parallel I/O on a shared file, recent studies [12, 16] have introduced the range lock instead of the mutex lock on an inode. The range lock provides exclusive or shared access to a specific range of a file. In other words, a thread is only blocked when there is another thread guaranteed exclusive access to an overlapping file range. Figure 3 shows an example of how the range lock works with parallel I/O to a shared file. Consider three threads, T_1, T_2, and T_3, writing a shared file in order. With inode mutex, all three threads will be serialized. On the other hand, the range lock allows parallel I/O from threads with non-overlapping ranges in the file as follows.

(1) T_1 starts writing its file range. Because no other threads are performing I/O, it is guaranteed that there is no range overlapping. At this moment, T_1 acquires a lock on its file range.
(2) T_2 first checks whether there is any range overlapping with its file range. There is a range that T_1 is writing, but it does not overlap with the file range T_2 is writing. Therefore T_2 can perform I/O and acquire a lock on its file range.
(3) T_3 also checks whether there is any overlapping range. However, unlike the case of T_1 or T_2, T_2 is performing I/O on a file range overlapping T_3's range. Thus, T_3 is blocked until T_2 completes its I/O and releases the lock on that range. Once T_3 makes sure that there is no overlapping range, it will perform the I/O by acquiring a lock on its file range.

3 PROBLEM STATEMENT

In this section, we describe the motivation for our work using the interference problem that occurs when multiple threads perform shared file I/O in F2FS, and define the cascading tree lock problem of *Node Tree*.

3.1 Lack of Concurrency on File Metadata

To illustrate the concurrency problem in accessing file metadata, we classified how a single I/O request is handled in F2FS into four phases.

(1) **Phase 1:** The first step is *I/O Initialization*. In this phase, the I/O request is passed to the kernel and sanitized. The first phase includes the initialization of the file system-specific data structures.
(2) **Phase 2:** The second phase is *Block Address Mapping*. The file system determines a set of block addresses for the I/O request by traversing the file metadata. Specifically, F2FS traverses Nodes via NAT translation as described in Section 2 with Figure 2b.
(3) **Phase 3:** The third phase is *Device I/O*. The actual I/O to the device occurs in this phase. Since we target a fast NVMe SSD (Solid State Drive), this phase is processed mostly in parallel due to the device's internal parallelism.
(4) **Phase 4:** The fourth phase is *Wrap-up*. After Device I/O is finished, the file system checks whether *Device I/O* succeeded. It also frees the file system-specific data structure and sets appropriate flags for related kernel data structures such as page cache. Lastly, the completion of the I/O request is handed over to the caller.

Figure 4 shows a pipeline view of file I/O operations with different F2FS implementations. Figure 4a and 4b illustrate how concurrent I/O requests are processed in baseline F2FS and F2FS with

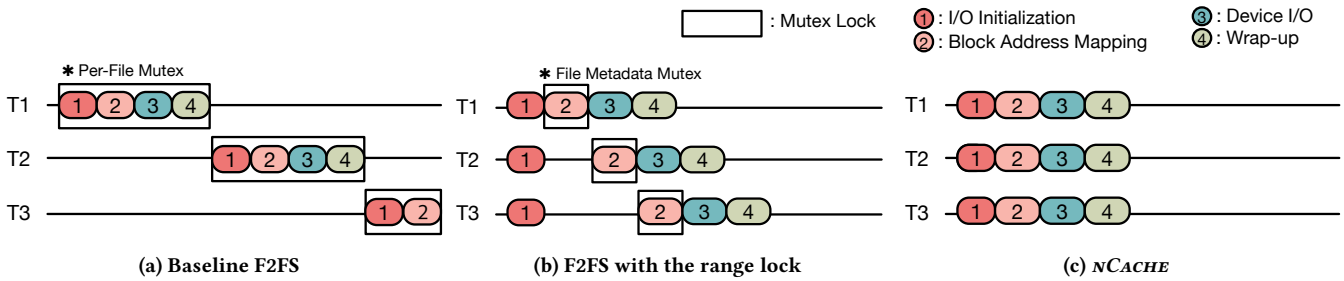


Figure 4: Pipeline view of parallel writes by multiple threads on a shared file.

range lock. In this scenario, there are three CPU cores, and one thread is running on each core. Each thread issues an I/O request to a shared file simultaneously, but none of the three I/O requests have overlapping ranges in the file. In the baseline F2FS (Figure 4a), all three I/O requests are serialized because of the per-file mutex lock in F2FS. Since the per-file mutex lock binds four phases into a single critical section, none of the phases can be pipelined with other threads.

On the other hand, F2FS with the range lock (Figure 4b) can pipeline some I/O phases such as I/O initialization, device I/O, and wrap-up phases because the range lock allows processing I/Os to non-overlapping ranges simultaneously. However, I/O requests are serialized again in the second phase (*Block Address Mapping* phase). Therefore, only limited parallelism can be exploited from the storage device. We have found that F2FS serializes the second phase due to the tree nature of the file metadata (*Node Tree*) in the page cache framework. We will elaborate on this in the following section.

3.2 Cascading Tree Lock in *Node Tree*

When F2FS performs *Block Address Mapping* (**Phase 2** in Section 3.1), it traverses *Node Tree* in a top-down manner to retrieve corresponding block addresses. While traversing the tree, F2FS holds two consecutive mutex locks until the end of the current traversal. In other words, whenever F2FS advances in *Node Tree* towards the leaf node, it has to acquire a mutex lock on the next child node before releasing the mutex lock of the current node. This is called the *cascading tree lock problem*, which causes severe lock contention when there are a large number of cores accessing *Node Tree* at the same time. Because every node (such as inode, direct and indirect nodes) in the tree requires a mutex lock to go down through that node, all threads get serialized at any node of the tree. The most trivial case of lock contention caused by cascading tree lock occurs at the root of *Node Tree*. Consider three I/O threads trying to perform *Block Address Mapping*. In this case, all of the threads have to acquire a mutex lock of the root node. Consequently, threads are serialized at the start of the *Block Address Mapping* phase. What makes the situation worse is that threads will compete again for the next mutex lock when they advance to the same child node.

Figure 4c shows a perfect pipeline view of parallel writes by three threads in the proposed nCache framework. They can access *Block Address Mapping* phase concurrently. The detailed design and implementation of nCache are described in the next section.

4 DESIGN AND IMPLEMENTATION

To solve lock contention in *Block Address Mapping* phase, we introduce nCache. Specifically, nCache employs readers-write lock (RW lock) for every node in *Node Tree*. The serialization in traversing *Node Tree* originated from the mutex lock in each node in the tree. With RW lock, nCache allows concurrent access in *Node Tree* and fully exploits the internal parallelism of the high-performance NVMe SSD device by performing all four steps of I/O requests in parallel. Next, we provide a design overview of nCache followed by a step-wise description of the nCache algorithm (Section 4.1 and 4.2). Furthermore, we identify and present the possible scenarios of inconsistent updates and discuss the consistent update method of nCache via double-checked locking (Section 4.3).

4.1 nCache Overview

Due to the *cascading tree lock problem*, *Block Address Mapping* phase is serialized at the traversal of *Node Tree*. To solve this problem, nCache uses RW lock for each node in the tree instead of mutex lock. Unlike baseline F2FS, nCache acquires the reader lock first for every advance in *Node Tree* traversal. Then, whenever a node in the tree needs to be updated, or a new node needs to be added, nCache releases the reader lock and reacquires the writer lock for the update.

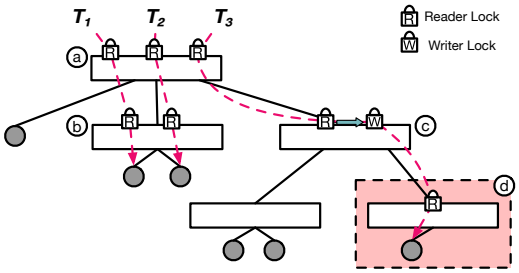


Figure 5: nCache inode structure.

Figure 5 shows an example of how multiple threads concurrently access in nCache. There are three threads accessing *Node Tree*. T_1 and T_2 want to retrieve block addresses of their own data blocks. In the meantime, T_3 wants to write at the end of the file. T_1, T_2, and T_3 share the root of the tree in their traversal. The three threads can read the root node concurrently because nCache always starts

by acquiring the reader lock (ⓐ). T_1 and T_2 share the child node again. However, both threads can read it without blocking since they are not modifying the node (ⓑ). Lastly, T_1 and T_2 retrieve corresponding block addresses and continue to the *Device I/O* phase. In the case of T_3, it starts with the reader lock at the child node of the root (ⓒ). Since T_3 wants to write at the end of the file, new nodes need to be added to *Node Tree* (ⓓ). Therefore, T_3 releases the reader lock and reacquires the writer lock on the current node. After T_3 is guaranteed exclusive accesses, new nodes are added into the tree. T_3 releases the writer lock as soon as the update is completed to minimize the blocking time of other threads. Finally, T_3 moves on to *Device I/O* phase.

With the range lock, *nCache* can provide finer-grained synchronization to perform *Block Address Mapping* phase in parallel. *nCache* blocks only when there is an update in *Node Tree*. In other words, blocking occurs only when there is a thread entered with the exclusive range lock. Since the range lock does not allow overlapping ranges to have exclusive access at the same time, blocking between writer locks occurs only when two threads share the last level of *Node Tree* except for the leaf node. In the case of blocking between reader lock and writer lock, blocking can be minimized since *nCache* only holds writer lock for a short period for adding a link to the new nodes in the tree. Figure 4c shows how *nCache* processes parallel I/O requests. As *Block Address Mapping* phase can be performed in parallel, all four phases are now not serialized and the parallel I/O is able to fully exploit the parallelism of the storage device.

4.2 *nCache* Algorithm

To enable concurrent access to *Node Tree* while ensuring consistent updates, we carefully designed the locking algorithm, as shown in Algorithm 1. The algorithm determines the locking behavior in *Block Address Mapping* phase. The *nCache* algorithm starts with the Node ID of the root in *Node Tree* (Line 2). First of all, *nCache* acquires the reader lock on the root node (Line 4). Then *nCache* repeats the following steps until it reaches the last level of internal nodes in *Node Tree* (Line 3 and 5). The first step is to read the Node and find the next Node ID towards the child node based on the offset in the I/O request. After that, it checks whether the next Node ID exists (Line 6). If the Node ID does not exist and the I/O request is a write operation, it means that the new node has to be added to the tree (Line 7-16). Otherwise, *nCache* releases the reader lock for the current Node ID and acquires the reader lock for the next Node ID. Then, *nCache* repeats the loop by setting the next Node ID as the current Node ID (Line 18-21).

Once *nCache* notices that *Node Tree* needs new nodes to be added, *nCache* releases the reader lock on the current Node ID and tries to acquire the writer lock instead (Line 7 and 8). After acquiring the writer lock, *nCache* rechecks whether it still needs to add a new node by searching the next Node ID in the current node (Line 9). When *nCache* makes sure that the new node needs to be added, the new Node ID is issued and connected to the current node (Line 10-11). After connecting the new Node ID to the current node, the next Node ID is set to the newly issued Node ID (Line 12). If another competing thread has already added a new Node ID, *nCache* simply uses it for the next Node ID (Line 14). In the end,

Algorithm 1: *nCache* Step-wise Algorithmic Flow

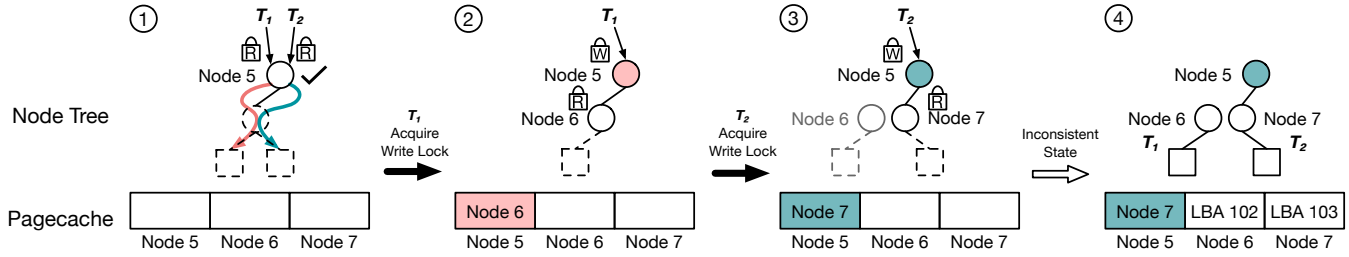
```
1  Algorithm nCache():
       Data: CurrentNodeID; NextNodeID
2      let CurrentNodeID be the Node ID of the root node in
         Node Tree;
3      let MaxLevel be the Maximum Level to retrieve the
         block address;
4      acquire read lock(CurrentNodeID);
5      while CurrentNodeID.level() <= MaxLevel do
6          if GetChildNodeIDByOffset(Node, Offset) == NULL
             and I/O is write then
7              release read lock(CurrentNodeID);
8              acquire write lock(CurrentNodeID);
9              if GetChildNodeIDByOffset(Node, Offset) ==
                 NULL then
10                 Get New Node ID;
11                 Record New Node ID to CurrentNodeID;
12                 NextNodeID = New Node ID;
13             else
14                 NextNodeID =
                     GetChildNodeIDByOffset(Node, Offset);
15             release write lock(CurrentNodeID);
16             acquire read lock(CurrentNodeID);
17         else
18             NextNodeID = GetChildNodeIDByOffset(Node,
                 Offset);
19         acquire read lock(NextNodeID);
20         release read lock(CurrentNodeID);
21         CurrentNodeID = NextNodeID;
22 Function GetChildNodeIDByOffset(Node, Offset):
23     if NodeID is exists at Offset in Node then
24         return NodeID;
25     else
26         return NULL;
```

the writer lock on the current Node ID will be turned back to the reader lock anywise (Line 15 and 16).

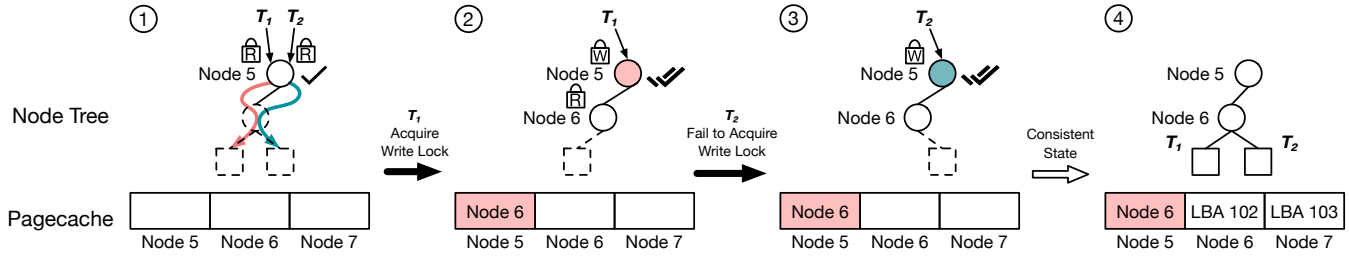
The reader lock is acquired for the next child node to continue *Node Tree* traversal, and the reader lock on the current node is released (Line 19 and 20). Finally, *nCache* moves on to the next child node and repeats the loop from the first step (Line 21).

4.3 Consistency in *Node Tree*

When an I/O thread wants to add a new node to *Node Tree*, the thread must acquire the writer lock on the current node after releasing the reader lock. However, reacquiring the writer lock might lead *Node Tree* to an inconsistent state. Figure 6a illustrates a possible scenario of an inconsistent update. Consider two threads T_1 and T_2 writing a new data block in disjoint ranges (①). At first, the two threads acquire the reader lock on the root node and check the Node ID of the child node. Due to the reader lock, both of the threads



(a) A possible inconsistent update scenario in the *nCache*.



(b) Consistent update with the Double-Checked Lock in the *nCache*.

Figure 6: Consistent update with the Double-Checked Lock in the *nCache*.

notice that there is no child node to advance and then decide to add a new node simultaneously. Thread T_1 and T_2 now compete to get the writer lock for the current node. Let's assume T_1 gets the lock first in this example (②). T_1 creates a new node (Node 6) and continues traversing the tree. After T_1 releases the write lock, T_2 gets the write lock. T_2 creates a new node (Node 7) and overwrites the link to a node T_1 created (③). As a result, Node 6 becomes unreachable, and the tree falls into an inconsistent state (④).

To provide consistent updates under the scenario mentioned above, *nCache* applies double-checked locking [19, 22, 23] in *Node Tree* traversal. Double-checked locking tests the lock condition first and acquires the lock if the condition is satisfied. After that, the lock condition is tested one more time whether the condition is still valid or not. By checking the condition twice before and after lock acquisition, *nCache* can avoid inconsistent updates. Figure 6b shows *nCache* with double-checked locking. As an example of an inconsistent scenario, consider two threads writing the file (①). When T_1 acquires the writer lock, it checks the existence of the child node again, and it creates a new node because the condition is valid (②). In T_2's turn, it also tests the condition once more. However, unlike in the inconsistent scenario, T_2 notices the child node T_1 already created and releases the write lock (③). Finally, T_2 safely traverses through Node 6 without making the *Node Tree* inconsistent (③).

5 EVALUATION

This section describes the experimental setup and analyzes the results of scalability experiments in F2FS with *nCache*.

5.1 Evaluation Setup

We performed all experiments on an eight-socket, 120-core (Intel Xeon E7-8870 v2 [7]) manycore server equipped with 740 GB of memory and three kinds of NVMe SSDs; Samsung 970 EVO [21], Intel SSD 750 [9], and Intel Optane 900P [8]. The detailed SSD characteristics are shown in Table 1.

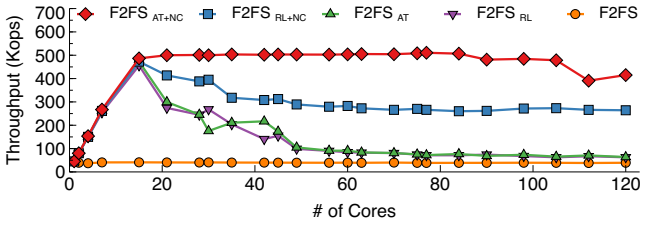
Implementation: We implemented *nCache* in F2FS in Linux kernel version 4.14. For range lock implementation, we used both an interval tree-based range lock [16] and an atomic operation-based range lock [12]. According to the design of the range lock, the atomic operation-based range lock had less lock contention than the interval tree-based design. For evaluations, we compared the following implementations:

- *F2FS*: Baseline F2FS.
- $F2FS_{RL}$: *F2FS* with interval tree-based range lock.
- $F2FS_{AT}$: *F2FS* with atomic operation-based range lock.
- $F2FS_{RL+NC}$: $F2FS_{RL}$ with *nCache*.
- $F2FS_{AT+NC}$: $F2FS_{AT}$ with *nCache*.

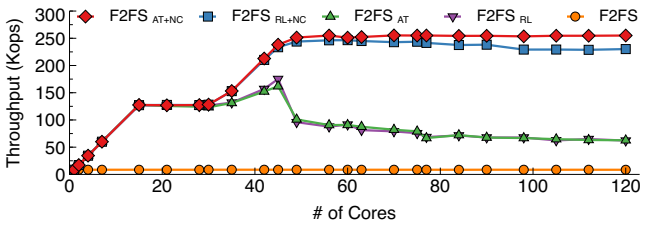
Workloads: We evaluated *nCache* using both synthetic and realistic workloads. For synthetic workloads, we used the FxMark [20] file system scalability benchmark. Among various workloads in FxMark, we selected DWOM and DRBM workloads to evaluate the benefits of *nCache* for parallel shared I/O workloads. Both workloads simulate a shared file I/O scenario. In the DWOM workload, each thread bound to a physical core writes 4 KB of data to a private region on a shared file. Since every private region does not overlap, it mimics the ideal parallel shared file write case. The DRBM workload works the same as the DWOM workload, but it reads 4 KB of data from the private region. In FxMark, each private

Table 1: Performance characteristics of various SSDs.

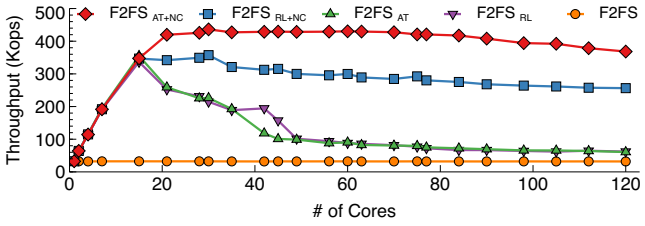
SSD	IOPS		Memory Type
Samsung 970 EVO [21]	200K	350K	NAND Flash Memory
Intel SSD 750 [9]	430K	230K	NAND Flash Memory
Intel Optane 900P [8]	550K	550K	Optane Memory



(a) Samsung 970 EVO

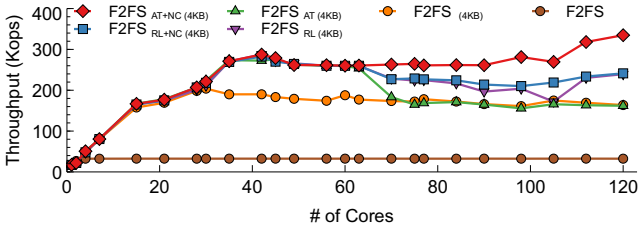


(b) Intel SSD 750

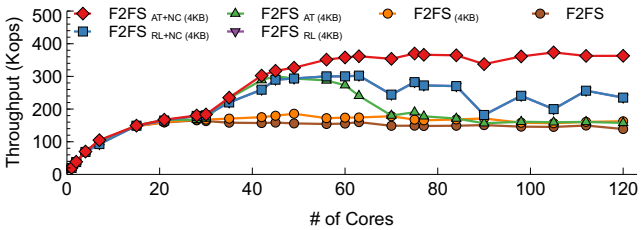


(c) Intel Optane 900P

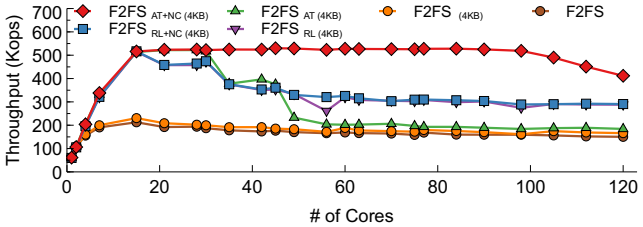
Figure 7: DWOM results for DirectIO.



(a) Samsung 970 EVO



(b) Intel SSD 750



(c) Intel Optane 900P

Figure 8: DRBM results for DirectIO. Specifically, both 8 MB and 4 KB stride sizes of the workloads were evaluated.

region is 8 MB, which means each thread issues I/O requests to the file with 8 MB strides. We measured I/O throughput by varying numbers of cores using both DirectIO and BufferedIO modes.

For realistic workloads, we used HACC-IO [26], which is an I/O benchmark for HACC [6]. HACC is a large cosmological simulation framework in the HPC environment. HACC-IO emulates the checkpoint phase of HACC. In HACC-IO, each MPI process writes simulation data to a different partition of a shared checkpoint file at the same time. We modified HACC-IO to use DirectIO mode to show how much *nCache* can saturate the storage device throughput. Another realistic workload we used is RocksDB [4]. RocksDB is a key-value store optimized for fast SSD devices and is widely employed as a storage engine in various databases and object storage systems [1]. We measured the throughput and latency by varying the number of cores issuing key-value operations to RocksDB. We used the *readrandomwriterandom* workload in *db_bench*, which is the benchmark shipped with RocksDB. In this workload, each core gets and puts random key-value pairs to RocksDB in a ratio of 9:1. We used RocksDB v6.14.5, and the rest of the tuning parameters of RocksDB were left as default.

5.2 Synthetic Workload Results

In this section, we evaluate and analyze the performance of Direct I/O and Buffered I/O for DOWM and DRBM workloads.

5.2.1 Shared File I/O Write (DWOM) using DirectIO. Figure 7 shows the results of parallel writes on a shared file using DirectIO. Due to inode mutex, *F2FS* does not scale and has the same throughput regardless of the number of cores in all three SSDs. On the other hand, all configurations with range locks show some level of scalability. However, the throughput of $F2FS_{RL}$ and $F2FS_{AT}$ collapses as the number of cores increases after a certain number of cores. With the range lock, only Samsung 970 EVO showed a device peak throughput at 15 cores but failed to sustain it (Figure 7a). As we mentioned in the early part of this paper, the maximum performance and parallelism of the fast NVMe device cannot be exploited solely by the range lock. $F2FS_{RL+NC}$ and $F2FS_{AT+NC}$ outperform other configurations. Specifically, $F2FS_{AT+NC}$ saturated the maximum device throughput in all storage devices, and the performance was sustained until 120 cores. Note that $F2FS_{RL+NC}$ had lower throughput compared to $F2FS_{AT+NC}$ due to the lock contention in the interval tree of $F2FS_{RL+NC}$. While $F2FS_{AT+NC}$ had the best performance among configurations in all SSDs, their scalability showed slightly different trends ($F2FS_{AT+NC}$ in Figure 7a, 7b, and 7c). We claim that this is because of the device's internal characteristics. For example, the peak performance of NAND flash-based devices can vary due to device-specific parameters such as NAND page size, internal garbage collection policy, or internal DRAM buffer.

5.2.2 Shared File I/O Read (DRBM) using DirectIO. Figure 8 shows the results of parallel reads on a shared file using Direct IO mode.

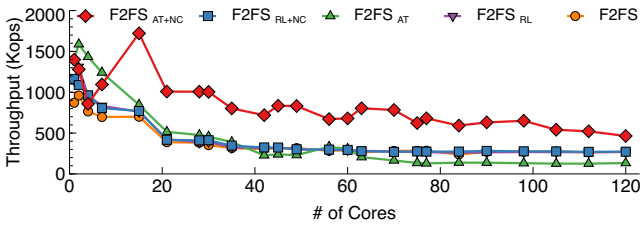


Figure 9: DWOM results for BufferedIO (Samsung 970 EVO).

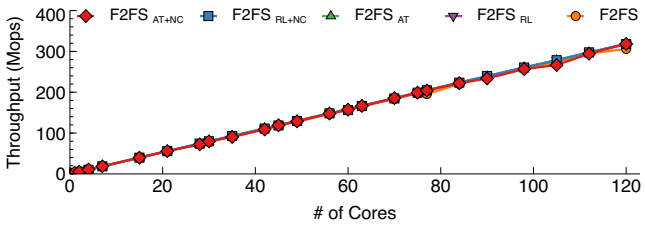


Figure 10: DRBM results for BufferedIO (Samsung 970 EVO).

Parallel reads are safe to concurrently traverse the *Node Tree* because they do not incur updates in *Node Tree*. However, in the baseline implementation, parallel reads are serialized at shared nodes in their *Node Tree* traversal path due to the *cascading tree lock problem*, as mentioned in Section 3.2. On the other hand, *nCache* allows concurrent reads traversing the *Node Tree* simultaneously because they share some node in the path using RW locks. However, with the DRBM workload, when I/O requests have an 8 MB stride, the SSD device throughput is largely degraded. In Figure 8a, 8b, and 8c, *F2FS* shows the throughput of the baseline configuration with the 8 MB stride of the workload. Basically, this low device throughput limits the performance and scalability of the baseline F2FS. With this workload (8 MB stride), none of the implementations showed a different trend from baseline performance. Thus, in the figures, we only show the baseline throughput since all configurations are bounded by low device performance.

To further explore the manycore scalability, we reduced the size of the private region to 4 KB (4 KB stride), which is the best case for the SSD, to identify the maximal throughput that *nCache* can sustain. Note that in the legend of Figure 8a, 8b, and 8c, 4 KB in parentheses means that the stride size is 4 KB. On the other hand, F2FS without parentheses means that the stride size is 8 KB. In Figure 8a and 8b, all configurations, including $F2FS_{(4KB)}$, have better throughput compared to *F2FS* until 30 cores. However, only $F2FS_{AT+NC\ (4KB)}$ achieves the maximal throughput of the device due to better concurrency in the file metadata by *nCache*. Note that the interval tree-based and atomic operation-based range lock designs show performance differences when *nCache* is applied. This performance gap between $F2FS_{RL+NC\ (4KB)}$ and $F2FS_{AT+NC\ (4KB)}$ shows that the concurrent access in the file metadata is inevitable to achieve scalability in the parallel I/O workload.

5.2.3 Shared File I/O Write (DWOM) using BufferedIO. We also evaluated the performance of *nCache* for the DWOM workload in the buffered I/O mode. Since all three SSDs display similar performance trends, we present results using only Samsung 970 EVO. In BufferedIO, F2FS has to acquire inode mutex to perform file write, the same as with DirectIO. As shown in Figure 9, range lock implementations outperform in the first 21 cores compared to *F2FS*. However, all implementations except $F2FS_{AT+NC}$ start to collapse at two cores, and the throughput of $F2FS_{AT+NC}$ also collapses at 15 cores. After that, only $F2FS_{AT+NC}$ sustains higher throughput than other implementations. The reason for the poor scalability in BufferedIO is a bit different from the case of DirectIO. In BufferedIO, the I/O to the storage device rarely happens because the page cache holds the user data in memory. Elimination of the I/O latency from the storage device imposes more contention on the range lock and RW lock in *nCache* than in the case of DirectIO mode. Additionally, F2FS has to reserve block addresses for future data flush from the page cache. This leads to more frequent node creations in *Node Tree*, and consequently, free Node IDs are consumed much faster. We observed that performance degradation is because of the serialization at the management of free Node ID.

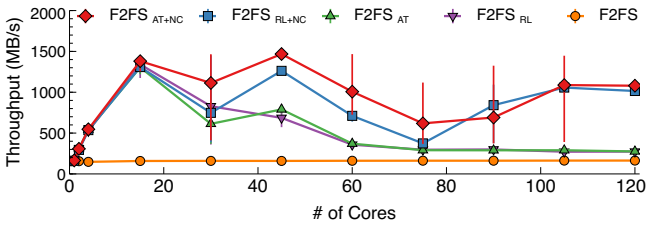
5.2.4 Shared File I/O Read (DRBM) using BufferedIO. Figure 10 shows the throughput of parallel reads to a shared file in the BufferedIO mode. In the DRBM workload, all implementations scale linearly until 120 cores. This is because read requests get the data by the page cache hit in most cases. When the up-to-date data is found in the page cache, the file system does not involve any of *Block Address Mapping* or *Device I/O* phases.

5.3 Realistic Workload Results

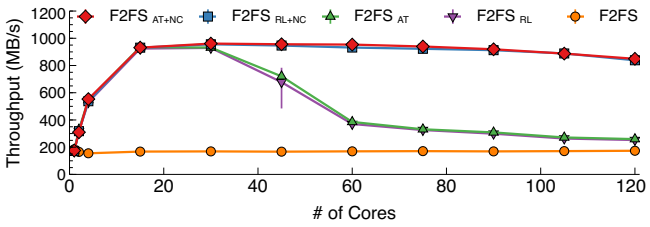
In this section, we evaluate and analyze the performance of realistic workloads, including the HPC scientific application (HACC-IO benchmark) and database application.

5.3.1 Scientific Application. Figure 11 shows the results of HACC-IO [26] using DirectIO mode. We fixed the size of the checkpoint as 10 GB and measured throughput by varying the number of MPI processes. Because of the per-file mutex lock in the file system, the performance of *F2FS* does not scale at all. On the other hand, $F2FS_{RL}$, $F2FS_{AT}$, $F2FS_{RL+NC}$, and $F2FS_{AT+NC}$ showed improvement in throughput. $F2FS_{RL}$ and $F2FS_{AT}$ exhibited similar trends due to the same reason from the DWOM workload evaluation results using DirectIO. $F2FS_{AT+NC}$ had higher throughput compared to other implementations. Specifically, in Figure 11a (Samsung 970 EVO), $F2FS_{AT+NC}$ showed 4.1x higher throughput compared to $F2FS_{AT}$ at 120 cores. Nevertheless, $F2FS_{AT+NC}$ had higher performance variation than $F2FS_{RL}$, $F2FS_{AT}$, and $F2FS_{RL+NC}$. In contrast, in Figure 11b (Intel 750 SSD) and 11c (Intel Optane 900P), they show little performance variation compared to Samsung 970 EVO. To find out the reason behind the high deviation of $F2FS_{AT+NC}$ with Samsung 970 EVO, we performed several experiments with different MPI process mappings. We observed that $F2FS_{AT+NC}$ sustains a peak throughput of ≈1400 MB/s, as shown in Figure 11a, when we bind cores from the first socket in order. In a Non-Uniform Memory Access (NUMA) architecture, both memory latency and bandwidth are degraded when a CPU accesses memory attached to a remote socket. We speculate that compared to other SSDs, the internal device characteristics of the Samsung 970 EVO are more affected by the characteristics of NUMA.

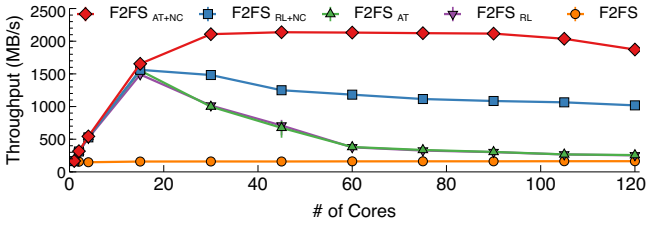
5.3.2 Database Application. Figure 12 shows the throughput and latency comparison of RocksDB using Intel Optane SSD 900P for



(a) Samsung 970 EVO



(b) Intel SSD 750



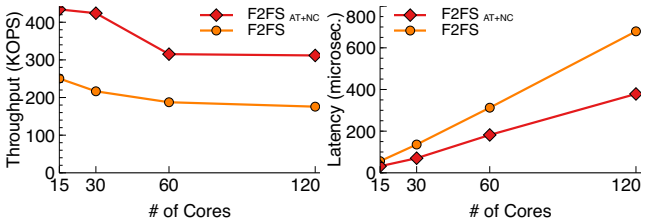
(c) Intel Optane 900P

Figure 11: Throughput of HACC-IO N-to-1 Checkpoint.

F2FS and $F2FS_{AT+NC}$. Each core produces 1 million operations with 16 bytes key and 100 bytes value. RocksDB is set to use DirectIO in file I/O. In Figure 12, $F2FS_{AT+NC}$ shows higher throughput and lower latency on average compared to *F2FS*. In Figure 12a, the performance gain in $F2FS_{AT+NC}$ compared to *F2FS* is mainly due to the improvement of file system lock contention by *nCache*. However, $F2FS_{AT+NC}$ fails to scale throughput with respect to the increased number of cores. This is because *nCache* has reduced the file system bottleneck as much as possible, but it is assumed that there is still a bottleneck in the DB application. Figure 12b shows a similar observation. The latency improvement of $F2FS_{AT+NC}$ over *F2FS* becomes more considerable for a large number of cores, but $F2FS_{AT+NC}$ fails to scale the latency due to the same reason from the throughput comparison.

6 RELATED WORK

There have been several studies on improving I/O performance by optimizing lock in HPC and distributed computing systems. In particular, a range lock allows multiple writes mutually to avoid serialization and reduce wait and delay in parallel I/O scenarios [2, 18]. Ching et al. [2] proposed a distributed lock manager for a file system with a byte-range locking scheme to optimize multiple write operations while ensuring consistency and atomicity for non-overlapped accesses to a shared file. Lin et al. [18] integrated a byte-range locking scheme into a cloud controller that manages concurrent accesses to non-overlapped regions of a shared file in



(a) RocksDB Throughput (b) RocksDB Latency

Figure 12: Performance Evaluation using RocksDB (Intel Optane 900P).

a distributed system. Both studies offer parallel I/Os of multiple threads simultaneously that access a shared file by using the range lock scheme while maintaining consistency. The parallel distributed file systems used in HPC such as Lustre [24] and Gluster [5] are implemented using range locks to improve parallel I/Os speed while preserving consistency and atomicity.

Recently, there has been an attempt to apply a range lock to the Linux native file system [12, 16]. In particular, Kim et al. [12] proposed an atomic operation-based range lock to minimize the overhead of implementing an interval tree-based range lock and applied it to a non-volatile memory-based file system (NOVA [27]). Also, inspired by an atomic operation-based range lock [12], Kogan et al. [14] proposed a scalable range lock that can be applied to general kernel memory structures. In this study, we found that there is a limit to increasing scalability in the Linux native file system by applying a range lock. Specifically, a range lock allows multiple threads to perform file data I/O in parallel when accessing a single shared file, but we claim that scalability is limited due to a bottleneck in the file metadata structure, not file data I/O.

7 CONCLUSION

Linux file systems have a scalability bottleneck when multiple threads perform I/O on a shared file. File I/O consists of file data I/O and file metadata I/O. File data I/O concurrency is provided by applying a range lock to the file system. However, even a file system employing a range lock fails to provide file metadata I/O concurrency. The main cause of the scalability bottleneck is the lack of concurrency in the file metadata structure. To mitigate this problem, we designed *nCache*, a novel file metadata cache framework that exploits the readers-write lock on a tree of the file metadata structure to allow concurrent accesses. Specifically, we implemented *nCache* in F2FS in Linux kernel version 4.14 and evaluated it with *nCache* on a 120-core manycore machine equipped with high-performance NVMe SSDs. Extensive evaluations showed that *nCache* saturates the maximum available I/O bandwidth of high-performance SSD devices while mitigating the file metadata structure bottleneck of F2FS. Moreover, from our investigation of various Linux file system architectures, we expect that *nCache* can be easily employed in other file systems due to the similarity of the inode structure.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System).

REFERENCES

[1] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[2] Avery Ching, Wei-keng Liao, Alok Choudhary, Robert Ross, and Lee Ward. 2007. Noncontiguous Locking Techniques for Parallel File Systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[3] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[4] Facebook. 2019. RocksDB. https://rocksdb.org/

[5] Gluster. 2019. Gluster File System. http://www.gluster.org/

[6] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, et al. 2016. HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. *New Astronomy* 42 (2016), 49–65.

[7] Intel. 2014. Intel Xeon Processor E7-8870 v2. https://ark.intel.com/content/www/us/en/ark/products/75255/intel-xeon-processor-e7-8870-v2-30m-cache-2-30-ghz.html

[8] Intel. 2019. Intel Optane 900P Series. https://ark.intel.com/content/www/us/en/ark/products/123628/intel-optane-ssd-900p-series-280gb-1-2-height-pcie-x4-20nm-3d-xpoint.html

[9] Intel. 2019. Intel SSD 750 Series. https://ark.intel.com/content/www/us/en/ark/products/86742/intel-ssd-750-series-400gb-2-5in-pcie-3-0-20nm-mlc.html

[10] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*. 24–35.

[11] Awais Khan, Taeuk Kim, Hyunki Byun, and Youngjae Kim. 2019. SciSpace: A Scientific Collaboration Workspace for Geo-Distributed HPC Data Centers. *Future Generation Computer Systems* 101 (2019), 398 – 409.

[12] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Changgyu Lee, Sungyong Park, and Youngjae Kim. 2019. pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. 1–7.

[13] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 691–706.

[14] Alex Kogan, Dave Dice, and Shady Issa. 2020. Scalable Range Locks for Scalable Address Spaces and Beyond. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. 1–15.

[15] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 273–286.

[16] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. 2019. Write Optimization of Log-Structured Flash File System for Parallel I/O on Manycore Servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*. 21–32.

[17] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 39–48.

[18] Yun Lin and Richard Sharpe. 2017. Using Byte-Range Locks to Manage Multiple Concurrent Accesses to A File in A Distributed Filesystem. US Patent 9,792,294.

[19] Scott Meyers and Andrei Alexandrescu. 2004. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal* (2004), 46–49.

[20] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. 71–85.

[21] Samsung. 2020. Samsung 970 EVO Series SSD. https://www.samsung.com/us/business/products/computing/ssd/client/970-evo-plus-250gb-mz-v7s250b-am/

[22] Douglas C Schmidt and Tim Harrison. 1997. Double-Checked Locking. *Pattern languages of program design* 3 (1997), 363–375.

[23] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. 2013. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Vol. 2. John Wiley & Sons.

[24] Philip Schwan. 2003. Lustre: Building a File System for 1000-node Clusters. In *Proceedings of the Linux symposium*. 380–386.

[25] Min Si, Antonio J Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. 2014. MT-MPI: Multithreaded MPI for Many-Core Environments. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*. 125–134.

[26] Venkatram Vishwanath. 2018. HACC I/O. https://github.com/glennklockwood/hacc-io

[27] Jian Xu and Steven Swanson. 2016. 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)*. 323–338.

[28] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 169–182.

[29] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220.

[30] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1629–1642.

[31] Yongen Yu, Douglas H Rudd, Zhiling Lan, Nickolay Y Gnedin, Andrey Kravtsov, and Jingjin Wu. 2012. Improving Parallel IO Performance of Cell-based AMR Cosmology Applications. In *Proceedings of the 26th IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS)*. 933–944.