IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 4, 2021, accepted January 15, 2021, date of publication January 27, 2021, date of current version February 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3054905

Parallelizing Shared File I/O Operations of NVM File System for Manycore Servers

JUNE-HYUNG KIM, YOUNGJAE KIM, (Member, IEEE), SAFDAR JAMIL, CHANG-GYU LEE, AND SUNGYONG PARK, (Member, IEEE)
Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea

Corresponding author: Sungyong Park (parksy@sogang.ac.kr)

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System).

ABSTRACT NOVA, a state-of-the-art non-volatile memory (NVM) file system, has limited performance due to its coarse-grained per-file lock when multiple threads perform I/Os to a shared file in a manycore environment. For instance, a writer lock blocks other threads attempting to access the same file, although they access different regions of a file. When multiple threads reading the same file share a cache line containing a reader counter, performance can be significantly degraded due to cache consistency protocol as we increase the number of readers. This paper proposes a fine-grained segment-based range lock (`SRL`) that divides a file into multiple segments and manages a lock variable dynamically for each segment. Consequently, write operations can be parallelized without blocking unless there is a conflict in accessing the same range in a file. Moreover, `SRL` maintains a reader counter per segment that allows multiple reader threads to perform read operations without causing a performance bottleneck. We evaluated an `SRL`-based NOVA on an Intel Optane DC persistent memory (PM) manycore server. The benchmarking results showed that the average write throughput of the `SRL`-based NOVA is 3× higher than the original NOVA, and the average read throughput scales linearly, while the original NOVA does not scale.

INDEX TERMS Operating system, file system, non-volatile memory, manycore CPU.

I. INTRODUCTION

In the last decade, distinct non-volatile memory (NVM) devices, such as phase-change memory (PCM) [1], resistive memory (ReRAM) [2] and 3D-Xpoint [3], have been studied and commercialized for various applications. Also, various NVM-based file system studies have been conducted while considering these NVM devices as primary storage [4]–[13]. These studies exploited the characteristics of NVM devices such as byte-addressability, low latency, and persistency, to provide high throughput and data consistency. Among these file systems, NOVA [4] is a state-of-the-art NVM-based file system that ensures the consistency of data and metadata in the event of power outage through its log-structured design. NOVA implements per-inode logging for metadata and uses per-core data structures such as inode table, journaling space, and memory allocator to boost performance by supporting concurrent I/Os [14].

However, we identified that NOVA does not fully exploit the scalability of I/O throughput when multiple I/Os are being performed to a shared file in a manycore environment. The major performance bottleneck is due to the coarse-grained per-file lock [15], [16], which negates the benefit of parallelism and high-performance NVM devices. Moreover, NOVA uses a counter-based readers-writer (RW) lock that allows multiple threads to read concurrently from a file while only one thread can write. This becomes another cause of performance bottleneck when the modification of a single shared RW lock variable invalidates the CPU cache lines of other read threads.

To mitigate the aforementioned problems, we have modified NOVA so that it adopts Linux's interval tree-based range lock [17], referred to as *range lock*. The interval tree is a well-known data structure for detecting overlapping ranges. The range lock first performs range checking for the I/Os before acquiring the locks. If ranges do not overlap, read and write operations can be performed, otherwise write operations are blocked. As the range lock controls read/write access to a range rather than a file, it is apparent that NOVA using a range lock should provide higher scalability in terms of I/O throughput than NOVA using a counter-based RW lock. However, we reported in our previous work [18] that the

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

NOVA using a range lock still does not scale for shared file I/Os as we increase the number of cores. This is because the current range lock implementation protects the entire tree with a mutex lock when inserting or deleting nodes in the tree. Therefore, many I/O threads have to compete for the mutex lock to access the interval tree on a manycore server, which serializes access and thus severely degrades I/O performance. Considering that the read/write latency of NVM is quite low, the software overhead caused by this lock contention along the I/O path has a significant impact on the execution time of read or write operations.

In addition, we proposed a fine-grained segment-based range lock (SRL) and apply it to NOVA to improve the scalability of shared file I/Os on manycore servers. SRL divides a file into fixed-size segments, where a segment is a contiguous set of pages in a file, and an RW lock is associated with each segment. In SRL, when read/write threads check overlapping ranges, they do not need to acquire an additional lock to protect the entire tree. The threads only check the segments corresponding to the I/O ranges to determine whether read-/write operations can be performed. Thus, SRL minimizes the serialization problem caused by range checking in the interval tree-based range locks. With SRL, when threads access segments, they run on their own CPU cores with their own RW lock variables. Thus, SRL can mitigate the performance degradation caused by the overhead from frequent CPU cache line invalidation.

SRL dynamically allocates an RW lock variable for each segment to efficiently utilize memory as the static allocation of all RW lock variables may result in inefficient usage of memory. For instance, if multiple threads read or write a small portion of a large file, it is a waste of memory space to keep the RW lock variables for all segments in memory. In order to dynamically manage RW lock variables, we adopted a radix tree where each leaf node represents an RW lock variable for each segment and the file offset is used as a key. In this case, a thread can lookup for its corresponding segment lock variable with its file offset through the radix tree, which allows multiple threads to access the tree concurrently.

On the other hand, when SRLs are employed in NOVA, the consistency problem attributed to NOVA's log-structured logging can occur. For example, when multiple threads attempt to write to a log concurrently by competing for a single log pointer, file system consistency can be damaged if it is not carefully managed. For this, we introduced a commit mark-based logging mechanism to ensure file system consistency. At the end of each write operation, NOVA commits the log entries by putting a commit mark on the entry along with the memory fence and cache line flush commands. In the recovery phase, only entries with commit marks become recoverable candidates. This commit mark-based logging mechanism safely completes the transactions from simultaneous multiple writers and makes the file system consistent after power failure.

We evaluated the scalability of SRL-based NOVA using FxMark [15] on a manycore server with 56 cores and Intel Optane DC persistent memory (PM) modules. In experiments using a shared file write workload, the write throughput of SRL-based NOVA is up to 3× higher than that of original NOVA and 28% higher than that of NOVA with the interval tree-based range lock. On the other hand, with a shared file read workload, the SRL-based NOVA scales linearly as the number of cores increases, while both original NOVA and NOVA with the interval tree-based range lock do not scale.

The contributions of this work are summarized as follows.

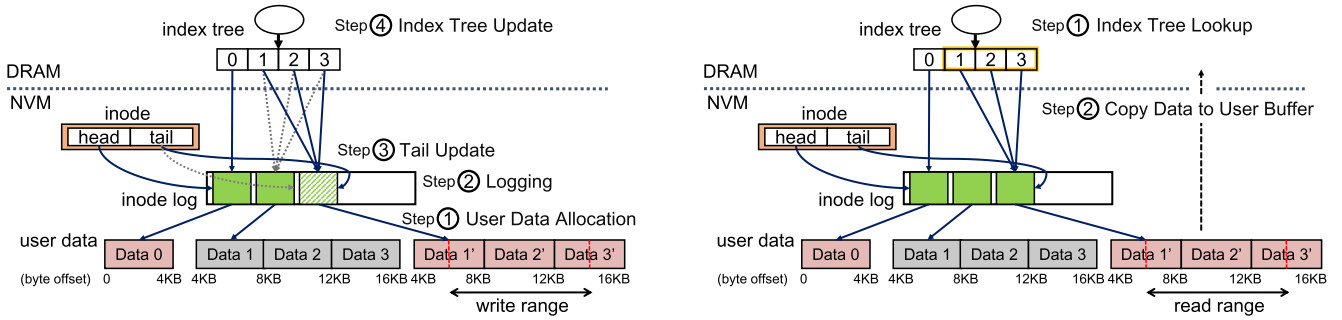
- We identified the scalability problem of shared file I/O in NOVA on manycore servers. In particular, we observed that there is a tree lock contention issue with the interval tree-based range locks.
- We proposed a fine-grained segment-based range lock called SRL that dynamically allocates RW lock variables for the segments to reduce the memory overhead.
- We proposed a commit mark-based logging mechanism to solve the file consistency problem caused by NOVA's log-structured logging.
- We modified NOVA by including various range locks and the commit mark-based logging mechanism. We also provided extensive evaluation results on a manycore machine with 56 CPU cores and Intel Optane DC PM modules, and showed that the SRL-based NOVA outperformed other NOVA versions.

II. BACKGROUND AND MOTIVATION

A. THE NOVA FILE SYSTEM

NOVA [4] is a state-of-the-art log-structured NVM-based file system. NOVA guarantees consistency of the file system using per-inode logging in NVM, where each log records every modification made to the files or directories. Specifically, NOVA uses the copy-on-write (COW) technique to update the file data where it first writes user data to newly allocated data pages and updates the pointer that points to the new data pages later. Figure 1 shows the operational behavior of NOVA and the core data structures such as index tree, inode structure, and per-inode log. NOVA maintains an index tree in DRAM for each file and uses it to retrieve the data pages corresponding to the file offset of a read or write operation. Each file has a 128 B inode in NVM, which includes head and tail pointers to the start and end of the log entries stored in the inode log. The inode log is a series of 4 KB log pages where the size of each log entry is 64 B and the user data is stored in 4 KB data pages. The log pages and data pages are allocated in NVM by a per-CPU memory page allocator.

Figure 1 also illustrates the read and write operations flow in NOVA. As shown in Figure 1(a), if a user wants to write data in the range of 6 KB-14 KB of a file, NOVA first allocates three new data pages as the unit of write operation is 4 KB page. It then copies the data (Data1, Data2, and Data3) to the newly allocated data pages by overwriting the user's data in the intended range of the data pages ①. After the data pages are completely written, the corresponding log entry is appended to the inode log. The tail pointer in an inode points



(a) Write I/O flow

(b) Read I/O flow

FIGURE 1. Write and read I/O flows in NOVA file system [4].

to the latest committed log entry and thus the new log entry is written right after the tail pointer. If the tail pointer points to the last log entry in a log page, a new log page is allocated and the log entry is written at the start of the new log page ②. Then, the tail pointer is updated to reflect the position of the new log entry ③. Finally, the index tree in DRAM is updated so that the new index node points to the new log entry ④, and NOVA's atomic write operation is finished.

The read operation in NOVA is relatively simple compared to the write operation. Figure 1(b) depicts the overall flow of the NOVA's read operation right after the write operation described above. When a read operation is initiated, NOVA first locates log entries corresponding to the range of read operation from the index tree ①. Since NOVA's write operation is atomic, the log entries found in the index tree point to the updated data pages. Thus, NOVA copies the data page to the user buffer and finishes the read operation ②. On the other hand, in the event of a sudden crash or power failure, NOVA can recover an inconsistent file into a consistent state by scanning the inode log. Beginning at the head pointer stored in the inode, all log entries are scanned until it reaches the position where the tail pointer is pointing. As a result, this scanning process can correctly reconstruct the file index tree and enables the recovery of an inconsistent file.

The write operation is atomic in the sense that the operation is either completely executed or not executed at all in the event of a crash or power failure. Therefore, if a crash occurs after user data is written without appending a log entry to the inode log, this operation is not visible because no log entry points to the user data. Even if a system crashes after writing a log entry but before updating the tail pointer, the operation is not also visible, since the log is located after the tail pointer and is excluded from the scanning process. The last case is when a system crashes after updating the tail pointer but before updating the index tree. In this case, because the log entry is pointed by the tail pointer, the scanning process includes the log entry, followed by an index tree update to that log entry. Therefore, users can access the newest data from the index tree, which means that the write operation is performed correctly.

B. MOTIVATION

In this section, we illustrate NOVA's scalability problems for shared I/Os and provide insights into a proposed solution by analyzing the experimental results of NOVA's scalability in a manycore environment.

1) SHARED FILE WRITER-WRITER PROBLEM

This problem occurs when multiple threads write to a shared file concurrently. The coarse-grained RW lock per file used in NOVA serializes writers, even if users write in different regions of a file. As shown in Figure 1(a), a NOVA write operation consists of the following four steps:

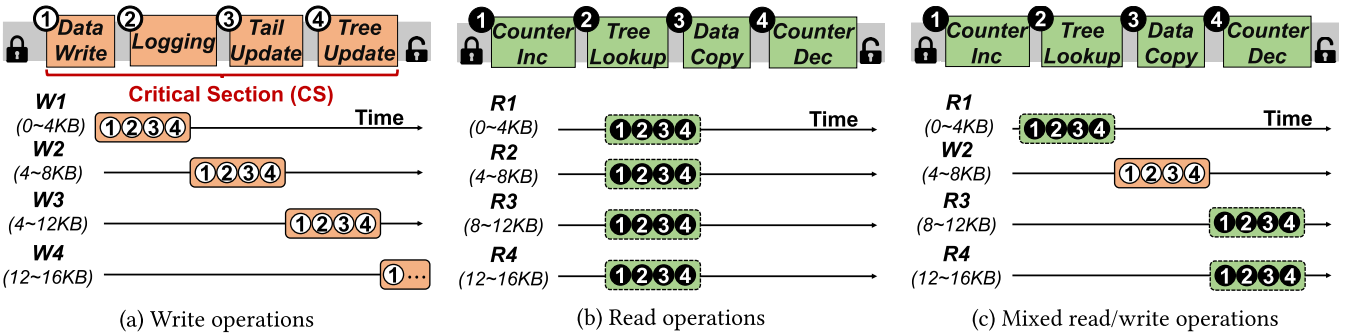
1) *Data Write:* Copy user data from user buffer to data pages.
2) *Logging:* Write a log entry containing information about a write operation in the inode log.
3) *Tail Update:* Update the tail pointer with a new log entry.
4) *Tree Update:* Update the index tree to point to the new log entry.

Figure 2(a) shows why a bottleneck occurs if four threads (*W1, W2, W3, W4*) write data to the same file simultaneously. In the current implementation, NOVA sets all the above steps as one large critical section. Therefore, a thread entering the critical section acquires a lock on a file by calling *down_write()* to prevent other threads from entering the critical section. After the thread completes the four steps, it exits the critical section by calling *up_write()* to release the lock. For example, *W1*-*W4* in Figure 2(a) can possibly execute the *Data Write* steps at the same time since they write data to different ranges of a file. However, in the current NOVA, once *W1* is in the critical section, all other threads should wait for *W1* to complete all four steps.

2) SHARED FILE READER-READER PROBLEM

A NOVA read operation, as shown in Figure 1(b), performs the following two steps:

1) *Tree Lookup:* Lookup log entries from the index tree corresponding to the range of the read operation.



(a) Write operations (b) Read operations (c) Mixed read/write operations

FIGURE 2. Pipeline procedure of concurrent shared file I/O operations in NOVA.

2) *Data Copy:* Copy user data from data pages to the user buffer.

Multiple threads can read data pages concurrently because it is not necessary to read data pages in a critical section. Figure 2(b) shows an example where four threads (*R1, R2, R3, and R4*) attempt to read from the same file simultaneously. However, NOVA uses the Linux RW lock and there is only one reader counter per file. NOVA first increases the reader counter before performing a read operation. When NOVA finishes the read operation, it decreases the reader counter. The update operation on the reader counter can be a bottleneck in the manycore servers if a large number of threads read from the shared file at the same time. This is because of the cache line invalidation overhead when multiple threads update a shared reader counter in the manycore servers.

3) SHARED FILE WRITER-READER PROBLEM

The NOVA per-file lock is implemented using the RW lock based on the reader counter mentioned above. That is, writers have to wait unless the reader counter is zero. Figure 2(c) shows an example where three reader threads (*R1, R3, R4*) and one writer thread (*W2*) execute their corresponding operations concurrently. In this case, two reader threads (*R3, R4*) have to wait while a writer thread (*W2*) is in the critical section because of the coarse-grained lock of a write operation.

4) SCALABILITY ANALYSIS FOR SHARED FILE I/OS IN NOVA

We evaluate the scalability of NOVA on the 56-core server with Intel Optane DC PM modules. We used the FxMark benchmark [15] to measure the bandwidth of NOVA for the following three workloads. The experimental setup is explained in detail in Section V.

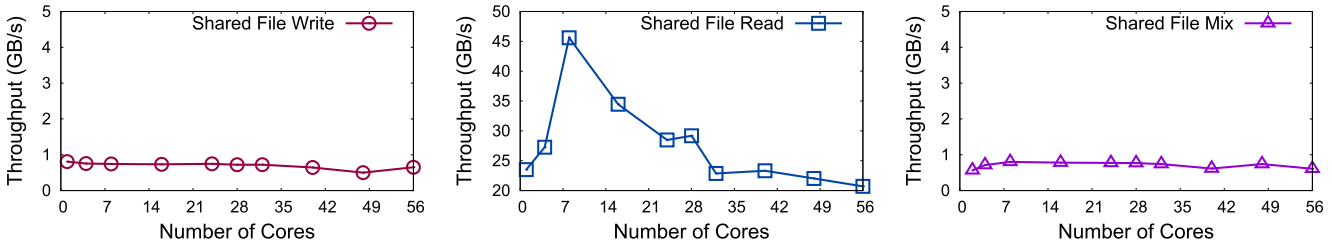
- *Shared file write:* each I/O thread only writes to the non-overlapping range of a shared file.
- *Shared file read:* each I/O thread reads only the non-overlapping range of a shared file.
- *Shared file mix:* each I/O thread reads or writes to the non-overlapping range of a shared file. The ratio of read threads to write threads is 50:50.

Figure 3 shows the results of scalability measurement of NOVA as we increase the number of I/O threads for the three workloads mentioned above. In these experiments, each I/O thread is pinned to a single core to minimize the thread scheduling overhead. Therefore, the number of threads in this paper is equal to the number of cores. Read/write sizes are set to 4KB. In Figure 3(a), we observed that the performance of NOVA does not scale at all as we increase the number of cores. Overall, the write throughput is under 1 GB/s regardless of the number of cores. This is due to the *shared file writer-writer problem*. The coarse-grained RW lock per file only allows a single thread to perform a write operation which serializes all the threads and concurrency among threads becomes impossible.

In Figure 3(b), we can see that the read throughput increases rapidly and reaches up to 45.5 GB/s on 8 cores. This is due to the CPU cache effect. In the shared file read workload, each thread repeatedly reads 4 KB pages. Thus, each operation can gain performance benefits from CPU cache hits. However, the throughput decreases after 8 cores. We conjecture that this is because the single shared reader counter becomes a performance bottleneck as discussed in the *shared file reader-reader problem*. The RW lock using a reader counter per file uses lock-free atomic operations to increase or decrease the reader counter. However, if many readers simultaneously modify the reader counter, the CPU hardware eventually serializes these accesses to the reader counter to maintain the correct value [19]. Also, the update operations on the reader counter invalidate the cache lines corresponding to the reader counter in other waiting threads and lock holders, which eventually degrades the overall throughput [20].

Figure 3(c) shows the results of using a shared file mixed workload. Similar to the results with the shared file write workload (i.e., 100% write), the performance of NOVA does not scale as we increase the number of cores. The measured throughput is under 1 GB/s regardless of the number of cores. This is because the total I/O throughput can be limited by the write throughput as explained in the *shared file writer-reader problem*.

For Figure 3, we ran a CPU analysis study. We found that the main reason for NOVA's non-scalable performance is the lack of available CPU cycles: when all of the CPU cores were used for I/O, the spinning operation to acquire the lock took about 95% of total CPU usage.



(a) 100% write-only workloads (b) 100% read-only workloads (c) 50% write, 50% read mixed workloads

FIGURE 3. Results of scalability measurement of NOVA file system as the number of cores performing I/O to shared files is increased.

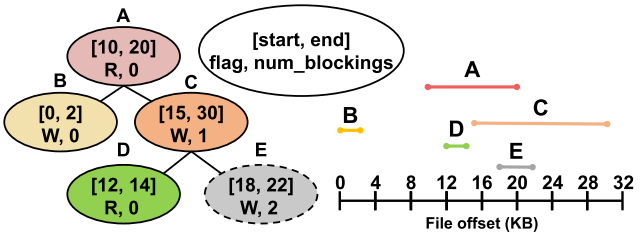


FIGURE 4. Example of interval tree and its operations for a file.

III. RANGE-BASED READERS-WRITER LOCK

The range-based RW lock selectively blocks I/O operations only if the overlapping ranges with a lock holder are accessed [21]. The interval tree-based range lock [17] is a range-based RW lock implementation for Linux that is distributed in the form of a Linux kernel patch. In this section, we explore the limitations of applying the interval tree-based range lock to the NOVA file system.

A. INTERVAL TREE-BASED RANGE LOCK

An interval tree is a well-known data structure that allows us to efficiently check whether any range from multiple I/Os overlap with a given interval. Figure 4 illustrates an interval tree and its operations for a file. As shown in Figure 4, a node in the tree contains information that represents the *start* and *end* of an interval. A node is inserted into the tree with its *start* value as a key for sorting. While being inserted, it executes an in-order traversal algorithm to find all overlapping intervals. The number of overlapping intervals found in this traversal is stored in the variable *num_blockings* of the inserted node. In the range-based locking mechanism, each interval is represented as a node in the interval tree. Since the RW lock allows reading of overlapping ranges between I/O threads, the reader checks the overlapping range only for the writer's node when inserting the tree node. For this, there is a flag on the tree node to distinguish whether it is a writer's node or not.

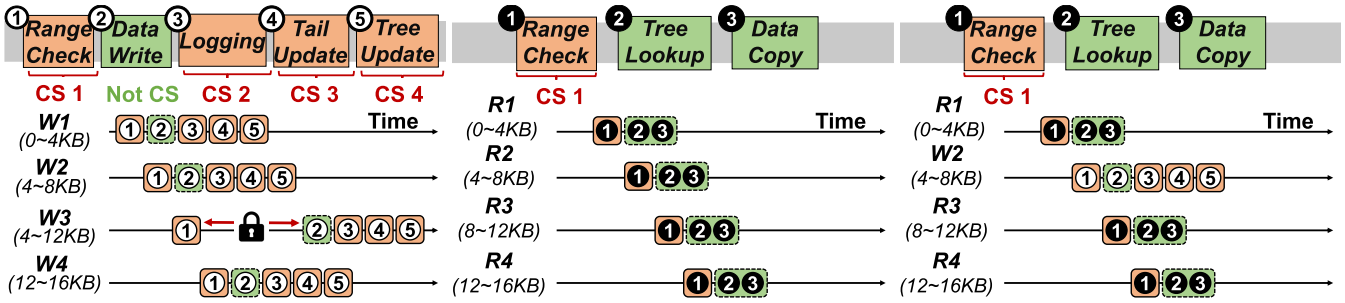
The process of inserting or deleting nodes in the interval tree is protected by a single mutex lock. For example, in Figure 4, assume that there are three range lock holders (*A*, *B* and *D* with *num_blockings* zero) and one range lock waiter (*C* with non-zero *num_blockings*). Also assume that a writer (*E*) with range *[18, 22]* is trying to acquire the range lock. To acquire the range lock, *E* locks the tree by holding mutex and finds all overlapping intervals by traversing the tree. Then, *E* stores the number of overlapping nodes to *num_blockings*. If *num_blockings* is zero, since it means that there is no overlapping lock holder, *E* adds itself to the node, releases the mutex and acquires the range lock. Otherwise, *E* cannot acquire the range lock since there is an overlapping lock holder(s). Therefore, *E* adds itself to the node, releases the mutex and blocks. Later, *E* is woken up when there is no overlapping lock holder.

B. APPLYING INTERVAL TREE-BASED RANGE LOCK IN NOVA

Figure 5 illustrates pipelining procedures of concurrent shared file I/Os in NOVA when the interval tree-based range lock is applied to NOVA. Figure 5(a) shows a case where multiple threads write to a shared file. As explained in Figure 2(a), all write steps are in the critical section and a writer can proceed only when its prior thread has finished the critical section. However, when range locking is applied, the large critical section can be divided into four smaller critical sections and all four threads can execute different steps in the pipeline.

As shown in Figure 5(a), the writer first performs *Range Check* ① to acquire a range lock. If there is no thread accessing the file range of the writer, the writer can perform the subsequent steps without waiting. Note that a node denoting the file range is to be inserted or deleted in the interval tree during range checking, and these operations must be protected via a mutex lock. Therefore, *Range Check* ① is a critical section. However, *Data Write* ② is not a critical section. Thanks to NOVA's per-core NVM page allocator, NVM pages can be allocated for each thread, and writes can be performed on the pages allocated independently. The next step is *Logging* ③, which is also a critical section. Even if writers append log entries to index different data pages, they must be serialized to determine the written position in the log page. *Tail Update* ④ and *Tree Update* ⑤ must also be protected by locks to guarantee the metadata consistency. As a result, steps ③, ④ and ⑤ are protected with different lock variables using the spinlock in the Linux kernel.

Figure 5(a) shows when four threads (*W1, W2, W3, W4*) race to write to a file. Unlike the case in Figure 2(a), even though the ranges of *W1* and *W2* do not overlap, *W2* has to



(a) Write operations (b) Read operations (c) Mixed read/write operations

FIGURE 5. Pipelining procedure of concurrent shared file I/Os in NOVA with interval tree-based range.

wait for *W1* to finish all the steps. However, *W2* does not have to wait *W1* to finish all the steps here, and it can execute when *W1* finishes the range checking step. If the ranges overlap, threads will be serialized by the range lock. For instance, as *W2* and *W3* have overlapping write ranges, *W3* must wait until *W2* finishes all the steps.

Figure 5(b) shows when four threads (*R1, R2, R3, R4*) read from a shared file. Readers also first perform the *Range Check* ❶, which is a critical section. If there is no writer in that range, readers can read the range of the file by adding the range node to the tree. Steps ❷ and ❸ are lock-free; thus, all four threads can execute these steps in parallel.

Figure 5(c) shows when one writer (*W2*) and three readers (*R1, R3, R4*) attempt to write to and read from a file, respectively, at the same time. Unless the writer overlaps with the ranges of the readers, the writer no longer serializes the readers and vice versa. Thus, unlike the case in Figure 2(c), *W2* can run in parallel with *R1*, *R3*, and *R4*.

C. SCALABILITY LIMITATION

The global lock used in the process of inserting nodes into or deleting them from an interval tree limits the scalability of NOVA on a manycore server. The reasons for this scalability limitation include the following:

First, the range checking step in the interval tree-based range lock where one single global lock is used for the entire tree becomes a bottleneck. When the interval tree-based range lock is used with existing block device (HDD and SSD) based file systems, the range checking process may not be a bottleneck because the speed of the storage media is relatively slow. However, in NVM, where the media is much faster and provides lower latency than HDD and SSD, the range checking step can largely affect the overall performance.

Second, readers with an interval tree-based range lock have to use a mutex lock on the tree. That is, while one reader checks the range of the interval tree, the other readers must wait, as shown in Figure 5(b). However, in the case of conventional NOVA, as shown in Figure 2(b), all readers can run in parallel. As a result, NOVA with the interval tree-based range lock shows relatively low parallelism due to the tree's mutex lock. Also, the competition for lock variables of mutex locks causes frequent cache line invalidation and coherence traffic, which is another factor in scalability limitation.

We have confirmed the two limitations in Section VI. In the next section, we describe a fine-grained range lock using hardware atomic operations to solve these limitations.

IV. FINE-GRAINED SEGMENT-BASED RANGE LOCK

A. SEGMENT LOCK VARIABLES AND MANAGEMENT

`SRL` is a fine-grained range lock, where each segment, a contiguous set of data pages, is managed by an RW lock. The segment size is a multiple of the 4KB page size. Each segment is assigned a 32-bit RW lock variable (*slv*). The left-most bit of *slv* is used for a writer lock. That is, if the segment is being written, this bit is 1 and otherwise the bit is 0. The remaining 31 bits are reserved for a reader counter, which represents the number of active readers for the segment. Suppose that a writer accesses a segment (*S*) of a file. It first checks the *slv* of *S*. If the left-most bit of the *slv* is 1, it has to wait because another thread is currently writing to the segment *S*. If the left-most bit of the *slv* is 0, it changes the bit to 1 using an atomic operation. Readers can be executed in parallel if there is no writer to *S* (i.e. the left-most bit of *slv* is 0). In this case, readers increase the reader counter using atomic operations. So, `SRL` can perform the *Range Check* step ① of the interval tree-based range lock in Figure 5 as lock-free. Other metadata operations in steps ③, ④ and ⑤ are still protected by locks.

We implemented `SRL` using a radix tree to manage *slv*s dynamically during execution. We call it `SRL`(tree). Figure 6 illustrates the design of `SRL`(tree). The radix tree is one of the well-known fast in-memory indexing data structures included in the Linux kernel. The internal nodes of the tree only contain an array of 2^n child pointers, while the *slv*s are pointed by leaf nodes. When a new file is created, the root node of the tree is created. Internal nodes and leaf nodes (*slv*) of the tree are dynamically allocated.

Figure 7 illustrates how to find the index of the corresponding *slv* for a file operation. It should be noted that the segment size, tree level, and index values are all configurable parameters. Suppose that the maximum file size provided by the file system is 1.5 TB, which is the maximum size of the PM disk partition in our Testbed (refer to Section V). Since

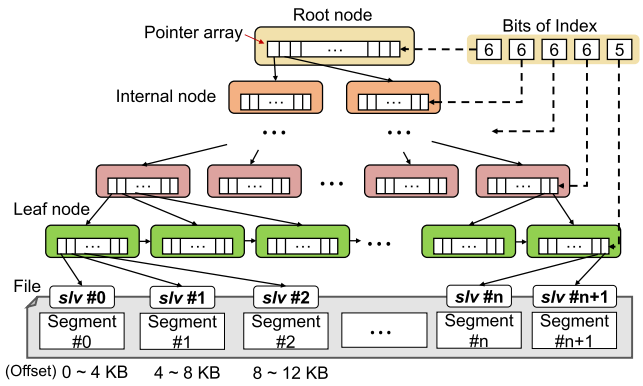


FIGURE 6. Description of the radix tree-based lock management for SRL(tree).

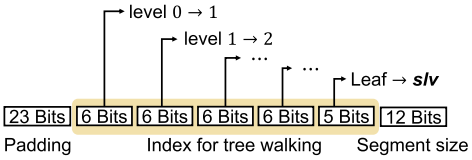


FIGURE 7. Translation of the file offset to find *slv* for the lock variable. Example of calculating the index of *slv* from the file offset accessed by the I/O thread.

the address space of 1.5 TB ($\approx 2^{41}$ bytes) can be represented by 41 bits, not all 64 bits for the file offset are used as an index. Considering that the segment size is 4 KB and *slv* is allocated per segment, the right-most 12 bits of the file offset are not used as an index and the other 29 bits are used for the *slv* index. In this example, the level of the tree is set to 5. Thus, for multi-level indexing, 6, 6, 6, 6, and 5 bits are used as index values for each level, respectively.

The pseudo-code for the operations of SRL(tree) is shown in Algorithm 1. The SRL(tree) operations have two steps. First, the *slv* corresponding to the file offset is located by indexing. SRL(tree) converts the file offset into index values to find the *slv* (line 10), which is shown in Figure 7. When traversing the tree from the root node to the leaf node using these index values, if the tree's internal node and *slv* are not allocated, they are dynamically allocated and inserted into the tree (lines 23∼38). Unlike the interval tree-based range lock, the insertion process of SRL(tree) does not use a mutex lock. Instead, it uses the function *atomic_cmpxchg*, which is a hardware atomic operation. The function changes the pointer of the array to the address of a newly allocated node only when the pointer is NULL. Since only one thread can update the pointer, the problem that multiple threads may change the pointer at the same time is avoided. The tree nodes and *slv* are de-allocated only when the file is closed. The de-allocation operation is performed only when the thread is the last user of the file by checking the number of processes opening the file. Meanwhile, an I/O operation can generally request multiple consecutive segments and lock them all together. For this, we slightly modified the radix tree so that it has leaf nodes connected in a linked-list fashion. We only need to do it once at the beginning of the tree search, since

```
1  typedef struct{
2      atomic_t *slv[max];
3  }SLVs
4
5  typedef struct{
6      int index[depth];
7  }PATH
8
9  void acquiring_SRL(inode,offset,len,write){
10     PATH path=translation_offset_to_path(offset);
11     SLVs target_slvs=searching_tree(inode,path,len
       );
12     if(write)
13         try_to_acquire_writer_lock(target_slvs);
14     else
15         try_to_acquire_readers_lock(target_slvs);
16 }
17
18 SLVs searching_tree(inode,path,len){
19     struct tree_node *cur=inode.SRL_root;
20     atomic_t *cur_slv;
21     SLVs corresponding_slvs;
22     /* Traversing tree through path and finding
       corresponding slv.*/
23     for(level=0;level<depth;level++){
24         child = path.index[level];
25         while(cur.arr[child] == NULL){
26             /*if the node or slv was not allocated
       */
27             new_node=allocate_new_node();
28                 /*or allocate_new_slv()*/
29             old=atomic_cmpxchg(&cur.arr[child],
30                                    NULL,new_node
       );
31             if(old != NULL)/*if failed to change*/
32                 free_node(new_node);
33             /*or free_slv(new_node)*/
34         }
35         if(cur.state == leaf_node)
36             cur_slv=cur.arr[child];
37         else
38             cur=cur.arr[child];
39     }
40     /*Calculate the number of segments to find,
41     i.e. the number of slvs, via len.*/
42     for(n=0;n<number_of_corresponding_slvs;n++){
43         corresponding_slvs[n]=cur_slv;
44         cur_slv=find_next_slv(cur,path);
45     }
46     return corresponding_slvs;
47 }
```

Algorithm 1. C Style Pseudo Code of SRL(tree)

the neighbor's *slv* can be searched following the linked list pointers (lines 42∼45). Second, SRL(tree) changes the value of the found *slv* using an atomic operation to perform an RW lock on the segment (lines 12∼15). The process of acquiring or releasing an RW lock with the corresponding *slv* that I/O threads find through the index tree is explained in the next section.

B. LOCK ACQUISITION AND RELEASE PROCESS

Algorithm 2 describes the lock acquisition and release process of the SRL. Suppose that a thread tries to acquire an RW lock from *start* to *end* segments. In this case, atomic operations supported by the hardware are used to acquire the RW lock for the segments. Note that the order of locking the segments is preserved to prevent deadlock. If a thread fails

```
1  void try_to_acquire_writer_lock(target_slvs){
2    unsigned int wlock = 1 << 31;
3    /* 10000...0, when write lock is held */
4
5    for(n=0;n<=number_of_corresponding_slvs;n++)
6      while (true) {
7        smp_mb_before_atomic(); // barrier
8        old=atomic_cmpxchg(&target_slvs[n],
9                           0, wlock);
10       smp_mb_after_atomic(); // barrier
11     /*write lock succeeds only
12                       when old slv was 0*/
13     /*namely, there is neither writer nor reader*/
14       if old == 0;
15         break;
16     }
17   return;
18 }
19
20 void try_to_acquire_readers_lock(target_slvs) {
21   unsigned int wlock = 1 << 31;
22   /* 10000...0 */
23
24   for (n=0;n<=number_of_corresponding_slvs;n++)
25     while (true) {
26       smp_mb_before_atomic(); // barrier
27       old = atomic_add_unless(&target_slvs[n],
28                               1, wlock);
29       smp_mb_after_atomic(); // barrier
30     /*if old slv was not wlock,
31                read lock always succeeds*/
32     /*namely, fail only when there is a writer*/
33       if old != wlock;
34         break;
35     }
36   return;
37 }
```

Algorithm 2. C Style Pseudo Code to Acquire and Release SRL

to acquire a lock on all segments, it cannot enter the critical section.

When a thread needs to write to a region from *start* to *end* segments, SRL attempts to acquire a writer lock by calling an *atomic_cmpxchg* function for each segment in [*start*, *end*] (lines 1∼18). If the old value of the *slv* is 0, which means that there is no writer or reader, the writer lock acquisition succeeds and the function changes the left-most bit of the *slv* to 1. Otherwise, the value of the *slv* is unaffected, and the writer lock acquisition attempt fails. Then, the writer thread will try to acquire the writer lock again.

On the other hand, a read thread tries to hold a reader lock by increasing the reader counter with the *atomic_add_unless* function which is also a hardware atomic operation (lines 20∼37). This function increases the value of the *slv* by 1 only when the left-most bit of the *slv* is not 1. This means that the reader lock fails only when the writer lock is already held.

To unlock the reader and writer locks, SRL does not require trying atomic operations repeatedly. In order for a writer to unlock, it simply clears the left-most bit using the *clear_bit* atomic operation. Since any writer or reader's lock acquisitions cannot succeed until the writer lock holder releases the lock, it is always safe to clear the left-most bit. Whereas, for a reader to unlock, it just decreases the reader counter by 1 using the *atomic_dec* atomic operation.

In the current implementation, when segment locks are held by readers, the priority of incoming readers is always higher than writers. In order to prevent writer starvation and provide fairness, an aging technique generally used by many RW locks [22] can be applied to SRL. If the waiting time for the writer to acquire the lock is too long, the writer can preempt the segment lock by temporarily blocking readers.

C. DYNAMIC SEGMENT LOCK MEMORY OVERHEAD

We can mathematically calculate the memory usage in SRL according to the setup variables (maximum file size, segment size, tree depth) and the size of file space actually used (working set).

For example, the index size (I) of each level (l) in the tree is defined in Equation 1, where M and S are the number of bits to represent the file size and segment size, and d represents the tree depth.

$$I \approx \left\lfloor \frac{M-S}{d} \right\rfloor \tag{1}$$

Then, the number of nodes required for each level (N_l) can be calculated as in Equation 2, where W is the number of bits to represent the working set size and $n = W-S-I*(d-l-1)$.

$$N_l = \begin{cases} 2^n & \text{if } n > 0 \\ 1 & \text{else} \end{cases} \tag{2}$$

Using the equations above, the total memory usage $F(x)$ in SRL can be calculated as Equation 3.

$$F(x) = \sum_{l=0}^{d-1} N_l * 2^I * \text{pointer size} + 2^{W-S} * \text{size of } slv \tag{3}$$

D. GUARANTEEING CONSISTENCY

The NOVA write operation updates the tail pointer after appending the log entry to the log block to validate the corresponding log entry. With the proposed range lock, NOVA increases write parallelism by allowing multiple threads to write data in parallel; however, in the end, concurrent write threads compete for the tail pointer where log entries are updated, resulting in a race condition. We solved this race condition by pre-determining the tail pointer location each thread uses before updating the tail pointer. In this way, each thread can update the tail pointer in their own location through atomic operations, eliminating the race condition on the tail pointer. The predetermination of the tail pointer location is protected by a lock and is mutually exclusive. In this process, if the current log page is full, log expansion occurs the same as NOVA.

The tail update, however, may lead the file system to an inconsistent state when multiple threads perform shared file writes concurrently. Suppose that two threads are performing a write operation to a shared file at the same time and the first thread completes appending its log entry before the second thread. Suppose also that a sudden power failure occurs while the first thread is trying to update the tail pointer after

TABLE 1. Intel Optane DC PM server.

CPU	Intel(R) Xeon(R) Platinum 8280M v2 2.70GHz CPU Nodes (#): 2, Cores per Node (#): 28
Memory	DRAMs per Node (#): 6, DDR4, 64 GB * 12 (=768GB)
PM	Intel Optane DC Persistent Memory PMs per Node (#): 6, 128 GB * 12 (=1.5TB)
OS	Linux kernel 4.13.0

the second thread finishes updating the tail pointer. The tail pointer now points to the log entry of the second thread. After the system is restarted, the file system is recovered by scanning log entries until it encounters the last entry to which the tail pointer points. Therefore, it is possible that the recovery process includes an invalid log entry such as that of the first thread, which leads to an inconsistent file system status.

To solve this problem, we introduced a commit mark-based mechanism to validate the log entry. After a log entry is appended, we add a commit mark to the entry. The commit mark write operation is done using atomic operations. Memory fence and cache line flush operations are also used to guarantee the memory order of the commit mark and log entry update. While recovering, the file system checks whether it is a vaild log entry with a commit mark to avoid inconsistency problems. Also, there may be valid log entries after the tail pointer, so we check the commit marks of all log entries in the file. Note that we are not parallelizing log entry insertions, but solving the inconsistency problem that can occur when writing file data concurrently.

V. EXPERIMENTAL ENVIRONMENT

To evaluate the scalability of NOVA, we conducted various experiments on a 56-core manycore server equipped with Intel Optane DC PM modules. The detailed specifications of the server are shown in Table 1.

A. SERVER ARCHITECTURE AND CONFIGURATION

The Intel Optane DC PM server is equipped with two sockets and 12 Intel Optane DC PM modules. Each socket has 28 processing cores and two on-board memory controllers (MC). One processor supports three memory channels per memory controller. The DRAM module and the PM module are paired and connected to one memory channel. Each memory module uses DDR4 (for DRAM) and DDR-T protocol (for PM) to communicate with the memory controller. Each socket is connected through ultra path interconnect (UPI) [23]. Processors that support the Optane DC PM modules provide PM store instructions such as *CLWB* to flush cache lines to PM and *non-temporal store* to bypass the cache hierarchy and write directly to the PM. The connection topology between processors and memory module is depicted in Figure 8.

The Intel Optane DC PM server provides three options for the use of PM modules.

- *Memory mode:* Optane DC PM modules lose data persistence and are used as extended DRAM memory.

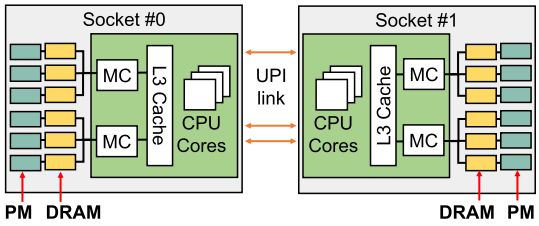


FIGURE 8. Processor structure for Intel Optane DC PM server.

- *Interleaved app direct mode:* Optane DC PM modules exist as persistent devices that are independent of DRAM. In this mode, the access to the PM is interleaved across six PM modules in a single socket. HW bottlenecks can be reduced by multiple threads accessing multiple modules in a striped manner, rather than multiple threads accessing a single PM module.
- *Non-interleaved app direct mode:* Similar to the interleaved app direct mode, Intel Optane DC PM modules are used as separate persistent devices. However, all access to the PM is directed to a single PM module.

In this study, the *interleaved app direct mode* was used.

B. MEMORY BANDWIDTH MEASUREMENT

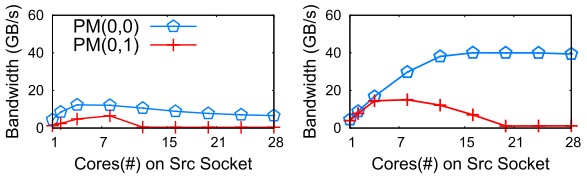
In this subsection, we evaluate the bandwidth of PM modules by increasing the number of cores for read only and write only workloads. For the experiments, we used the memory latency checker (MLC) tool by Intel [24]. The MLC tool measures peak bandwidth for workloads, in which multiple threads read or write sequentially to memory devices. In particular, for accurate memory latency measurements, the MLC tool disables the hardware pre-fetcher.

In the non-uniform memory access (NUMA) system, accessing the memory located at the remote socket has higher latency and lower bandwidth than accessing the local socket. In this paper, performance degradation caused by remote socket access compared to local socket access is called the *NUMA effect*. To accurately measure the PM performance considering the NUMA effect, we experimented with the following configuration.

- *PM(src,dst):* Cores performing I/O in socket *src* access the memory buffer allocated to Optane DC PM in socket *dst*, where *src* and *dst* are the socket numbers depicted in Figure 8.

Since PM(0,0) and PM(1,1) (also PM(0,1) and PM(1,0)) had identical performance, we ran experiments using just one of the two configurations. Memory buffers are allocated per core and are not shared between cores. The size of the buffer is 300 MB, which is large enough to cause L3 cache miss.

Figure 9(a) shows the write bandwidth measurements for PM(0,0) and PM(0,1) by increasing the number of cores. In this experiment, write threads use the 64-byte *non-temporal store* instruction, which is the same method as that used



(a) Write bandwidth (b) Read bandwidth

FIGURE 9. The results of write and read bandwidth measurements by increasing the number of cores in each memory module.

when writing data blocks in NOVA. As shown in Figure 9(a), the write bandwidth of PM(0,0) increases up to 12.3 GB/s on 4 cores and gradually decreases after 4 cores, whereas the maximum bandwidth of PM(0,1) is about 6.5 GB/s on 8 cores, and it also continues to decrease as we increase the number of cores.

It is assumed that the reduction of write bandwidth is due to the two hardware limitations of the Optane DC PM server described in [25]. First, data is stored in the write pending queue (WPQ) inside the memory controller before being loaded onto the data bus. Data that reaches the WPQ through PM store instructions such as *CLWB* and *non-temporal store* are guaranteed to be flushed to the Optane DC PM module even if a power failure occurs (within the hold-up time). Therefore, the contention of the queue can adversely affect the write bandwidth. Second, the data arriving at the Optane DC PM module is stored in the combining buffer. Contention for space in the buffer will lead to performance degradation with higher thread counts. Interestingly, the write bandwidth of PM(0,1) degrades more rapidly than that of PM(0,0) as we increase the number of threads performing concurrent writes. The write bandwidth of PM(0,0) on 28 cores is about 6.6 GB/s, while the write bandwidth of PM(0,1) is only 0.4 GB/s. This is also because of the hardware limitation explained in [25] and the NUMA effect.

Figure 9(b) shows the read bandwidth measurements for PM(0,0) and PM(0,1) by increasing the number of cores and performing 64-byte read operations from the buffer in the target PM. As shown in Figure 9(b), the read bandwidth of PM(0,0) increases linearly up to 40.04 GB/s on 28 cores as we increase the number of cores. While the bandwidth of PM(0,1) also increases up to 15 GB/s on 8 cores, it starts to decrease after 8 cores due to the same reasons explained in Figure 9(a).

In summary, we make the following observations from Figure 9. First, the read bandwidth is generally higher than the write bandwidth in PM. For instance, the read bandwidths of PM(0,0) and PM(0,1) are 3.24× and 2.31× higher than the corresponding write bandwidths, respectively. Second, the NUMA effect in PM is more significant in performing write operations than read operations, as we increase the number of cores. That is, the write bandwidths of PM(0,0) are 1.86× and 16.5× higher than those of PM(0,1) on 8 cores and 28 cores, respectively. However, the read bandwidths of PM(0,0) are 1.98× and 35.7× higher than those of PM(0,1) on 8 cores and 28 cores, respectively.

VI. EXPERIMENTAL RESULTS

A. EXPERIMENTAL SETUP

Baselines: In this section, we evaluate `SRL` on the Testbed described in Section V. For the experiments, we made four versions of NOVA.

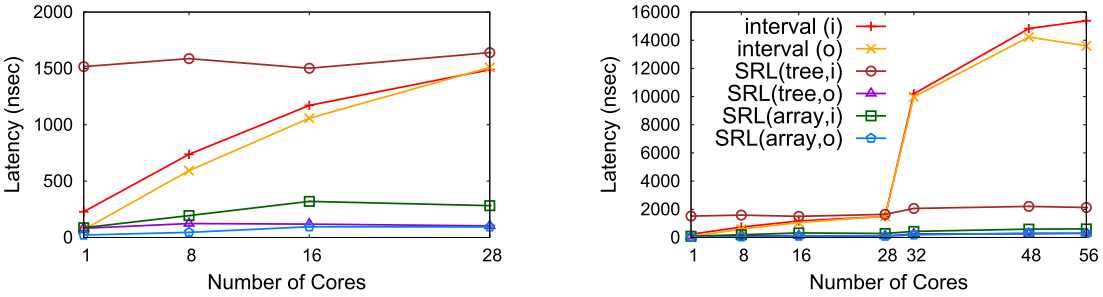
- *NOVA-default*: Vanilla NOVA using an RW lock in Linux.
- *NOVA-interval*: NOVA using an interval tree-based range lock in Linux.
- *NOVA-*`SRL`*(array)*: NOVA using an array-based implementation of `SRL`. The size of the *slv* array is fixed to the maximum size allowed by the file system [18].
- *NOVA-*`SRL`*(tree)*: NOVA using a radix tree-based implementation of `SRL`, where *slv*s are dynamically allocated and freed.

In particular, to study the space efficiency of `SRL`, we intentionally compared static (array) and dynamic (tree) versions of `SRL` in our experiments. When indexing *slv* into an array, we can predict the most ideal performance with NOVA-`SRL`(array). However, there is the problem of wasted space with this approach. For `SRL`, the default segment size is 4 KB in the experiments unless the size is specifically mentioned. In addition, the level of the radix tree is 5 in `SRL`(tree) and the index of each level is divided as shown in Figure 7.

PM Device Configuration: We created a PM device using the PM modules on socket 0 with an interleaved app direct mode because the PM modules across sockets cannot be aggregated as a single device in the current Testbed.

Workloads: For synthetic workloads, we used the FxMark benchmark [15] for evaluations that can generate various workload patterns. Specifically, we used shared file I/O workloads such as DWOM and DRBM. The DWOM workload performs multi-threaded shared file writes where multiple writers write to a shared file. The DRBM workload performs multi-threaded shared file reads where multiple readers read from the same file. We also modified the FxMark to mix the aforementioned two workloads where the read and write ratio of the workload patterns can be adjusted. In each workload, each thread continuously issues write/read requests to mutually exclusive regions of a file between threads. We experimented with each workload several times to get the average value and error rate using the standard deviation. When performing benchmarks, I/O threads and CPU cores are mapped one-to-one. In addition, FxMark pre-allocates files with the size of the number of cores×8 MB before measuring performance.

For realistic workloads, we used HACC-IO [26] and RocksDB [27]. HACC-IO is an I/O benchmark for HACC [28], a cosmological simulation framework for an HPC environment. In HACC-IO, each MPI process simultaneously writes data to different partitions of a 10 GB shared checkpoint file. RocksDB is a high-performance NoSQL database based on log-structure merge (LSM) trees. We experimented with the scalability of NOVA using *DBbench*



(a) Latency over a single socket - 28 cores (zoom).

(b) Latency over two sockets - 56 cores.

FIGURE 10. Measurements of the maximum lock acquisition latency for acquiring three versions of range locks.

in RocksDB, which is commonly used to measure file system performance. In our RocksDB experimental setup, several threads simultaneously generate random-read and random-write for 10 M keys and 100 B values.

B. PERFORMANCE EVALUATION

1) LOCK ACQUISITION LATENCY

Figure 10 compares the maximum latency measurements of three range locks (i.e., interval tree-based range lock, SRL(array), and SRL(tree)) by increasing the number of threads trying to acquire a lock. In this experiment, we measured the range lock acquisition latencies for two writing patterns such as initial writing and overwriting using the DWOM workload. For example, the results with legend (i) such as interval(i), SRL(tree,i) and SRL(array,i) indicate the latency measurements when the initial writing operations to a file are performed, whereas the results with legend (o) are the latency measurements for the overwriting operations.

Figure 10(a) depicts the latencies measured over a single socket (i.e. up to 28 cores). As shown in Figure 10(a), the latency of SRL(array) is relatively low compared to those of other range locks regardless of the number of cores or writing patterns, because the lock variables are statically allocated. On the other hand, in the interval tree-based range lock where a coarse-grained mutex lock is necessary to insert a range node into the tree, the latency increases almost linearly as we increase the number of cores. That is due to the contention for the mutex lock as the number of lock competitors increases. As a result, the lock acquisition latencies of interval(i) and interval(o) are 5.2× and 16.2× higher than those of SRL(array) on 28 cores. In SRL(tree), the latency varies depending on the writing patterns rather than the number of cores. The latency of SRL(tree,i) is less affected by the number of lock competitors since mutex locks are not used for the index trees. However, SRL(tree,i) has the highest latency of 1,500 nsec on 1 core because *slv*s need to be dynamically allocated when threads initially write to a file. In contrast, SRL(tree,o) has low latency similar to SRL(array) because *slv*s are already allocated when the overwriting operations are performed.

We also checked the lock acquisition latencies of range locks across two sockets (i.e., up to 56 cores) to see whether the NUMA configuration affects the overall latency. As shown in Figure 10(b), the latencies of interval(i) and interval(o) rose sharply after 28 cores, which is the NUMA boundary of our Testbed. For example, the latencies on 56 cores are almost 10.3× and 9.02× higher than those on 28 cores. This can be explained by the following reasons. In the interval tree-based range lock, each thread is supposed to modify the lock variable for the mutex lock, which is shared by all threads participating for the race condition. Then, the modification of this shared variable results in invalidating the cache lines including the lock variable of waiting threads and even lock holders. In particular, when a cache line is shared among threads in a NUMA environment, frequent invocations of the cache line invalidation to maintain the cache coherence become a significant overhead [19], [29]. In the case of SRL(tree,i), when it dynamically allocates *slv*s, the cache line for atomic operations can be shared. Thus, the latency on 56 cores is 1.29× higher than that on 28 cores. On the other hand, this overhead does not exist in SRL(tree,o) and SRL(array), because the lock variables for the I/O threads are already assigned and are not shared.

There is also an overhead of SRL for a single thread. For example, for large-sized write operations, NOVA only takes a single lock, whereas SRL has to lock multiple segments. Nevertheless, the overhead of SRL has little effect on overall performance. In NOVA-SRL, the total time taken when a thread executes a write operation is expressed as follows.

$$T(\text{total}) = T(\text{page write}) + T(\text{lock acquisition}) \quad (4)$$

At this time, T(lock acquisition) which is the time to acquire locks for all segments, is very small compared to T(page write) which is the time to write to the PM. So the total execution time is bound to T(page write). In our experiment, the time it takes for one thread to write 1 GB was 1.42 sec and 1.44 sec for NOVA-default and NOVA-SRL(tree), respectively. As a result, NOVA-SRL can provide concurrency by parallelizing multiple I/O operations, while it does not have much performance overhead compared to NOVA, even for a large single I/O operation.

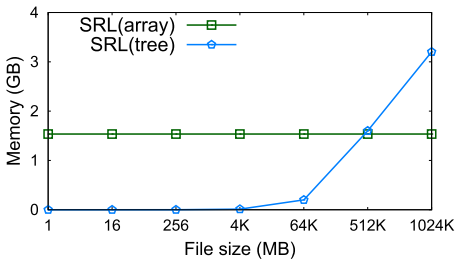


FIGURE 11. Comparison of additional memory usage for segment management.

2) MEMORY USAGE ANALYSIS

We evaluated the memory usage of SRL based on Equation 3 discussed in Section IV-C. For fair evaluation, we assumed that the maximum file size in the file system is 1.5 TB, and each segment size is 4 KB.

Since the actual size of a file is not known in advance when the file is created, SRL(array) statically allocates the number of *slv*s based on the maximum file size regardless of the actual file size. Thus, estimating by substituting the working set as the maximum file size, SRL(array) requires about 1.5 GB of memory space as shown in Figure 11. On the other hand, SRL(tree) uses a memory space proportional to the actual file size in use. When the file size is smaller than 512 GB, SRL(tree) always uses less memory than SRL(array). It is worth noting that the amount of memory used for the interval tree-based range lock increases proportionally to the number of I/O threads. If it is assumed that the node size of the tree is 128 B and the total number of threads performing file I/O (i.e., the number of threads actually writing or reading the file + the number of threads waiting to acquire locks) is N, the memory usage of the interval tree-based range lock will be $N * 128$ B. However, the memory use of SRL(tree) and SRL(array) is only affected by the file size, not the number of threads.

3) SCALABILITY ANALYSIS

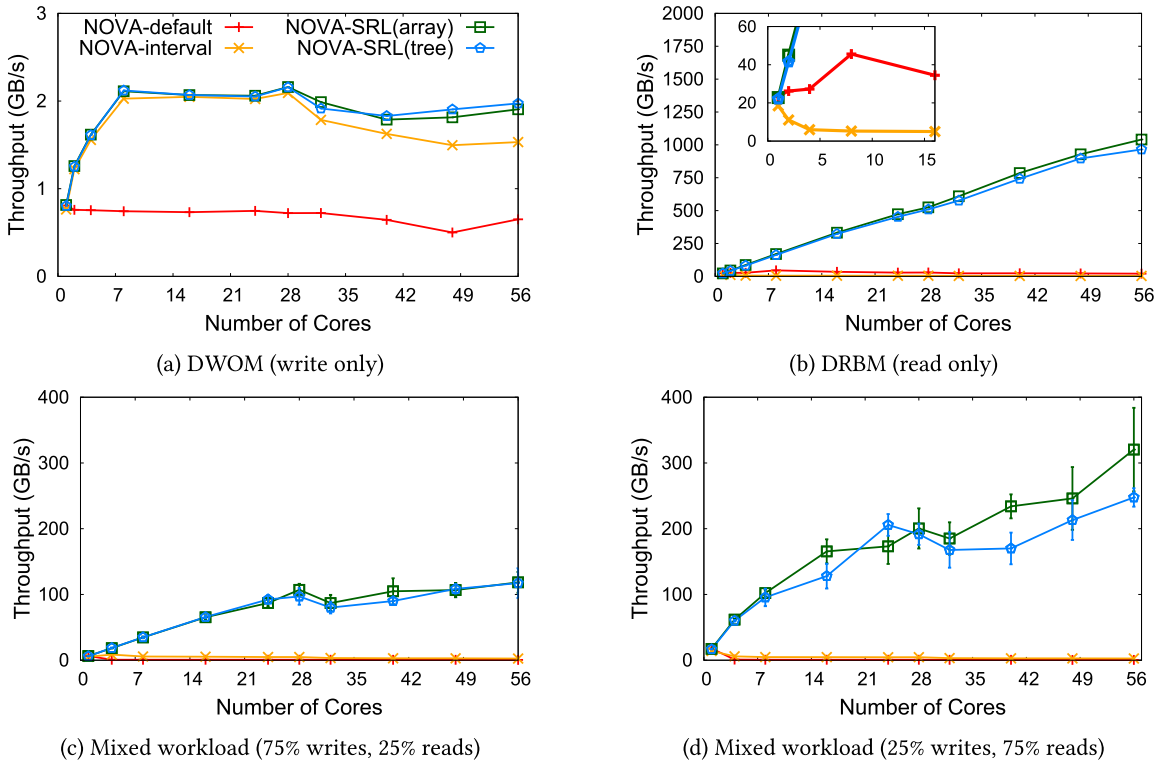
Figure 12 analyzes the scalability of four NOVA versions with different I/O patterns. As shown in Figure 12(a), NOVA-default is not scalable in the DWOM workload due to the shared file writer-writer problem. NOVA-default offers only 0.8 GB/s on 1 core and its performance slightly decreases afterward. On the other hand, the maximum throughputs of other range lock-based NOVA versions reach up to 2 GB/s on 8 cores and decrease after 28 cores. The throughput stalling after 8 cores is because of the congestion in the memory controller's WPQ (as explained in Figure 9(a)). Also, the reason why the throughput begins to decrease after 28 cores is due to the NUMA effect. We also observed that the decreasing rate of NOVA-interval is greater than those of NOVA-SRL(array) and NOVA-SRL(tree). This is because the mutex lock used for the interval tree is not *NUMA-aware*, as seen in the previous section. Finally, the throughput of NOVA-SRL(tree) is very similar to that of NOVA-SRL(array) because DWOM is a workload that overwrites blocks and the overhead for the index tree in NOVA-SRL(tree) is negligible.

Figure 12(b) shows the throughput results of four NOVA versions using the DRBM workload. In NOVA-default, the throughput reaches up to the maximum 45.5 GB/s on 8 cores, and subsequently decreases to 20.7 GB/s on 56 cores. This is attributed to the use of a reader counter lock in NOVA-default. That is, multiple readers can increase the reader counter while allowing read operations in parallel. However, as more threads attempt to modify the shared reader counter, the cache line invalidation overhead severely increases and the throughput begins to drop. Interestingly, the throughput of NOVA-interval is only 18.4 GB/s on 1 core, and it continues to decline afterward to 3.2 GB/s on 56 cores, which is far worse than that of NOVA-default. For every read operation, the reader first holds the coarse-grained tree lock, inserts itself into the tree, traverses the tree to find overlapping nodes, and then releases the lock. This coarse-grained tree lock completely serializes the reader lock acquisition process and negates any benefit of using multiple threads. NOVA-SRL(tree) and NOVA-SRL(array) show great scalability up to 56 cores. Since RW lock variables are defined for each segment of a file, the reader lock acquisition does not invalidate the cache line. Thus, there does not exist any serialization point in acquiring the reader lock. Also, in the DRBM workload, since each thread reads the same data repeatedly, cache misses for the data can barely occur. This leads to substantially high throughput up to nearly 1 TB/s. For instance, the throughput of NOVA-SRL(array) is 1,040 GB/s, while that of NOVA-SRL(tree) is 965.3 GB/s on 56 cores. It can be seen that NOVA-SRL(tree) has slightly lower performance on 56 cores due to its tree-traversing overhead.

Figure 12(c) shows throughput results using the mixed workload with a 3:1 write and read ratio. In this case, the throughput of NOVA-default does not scale at all because of the low write throughput. Similarly, NOVA-interval also does not scale due to its poor read performance despite the benefit of parallel writes. On the other hand, NOVA-SRL(array) and NOVA-SRL(tree) have scalable performance and their throughputs reach around 120 GB/s on 56 cores. Figure 12(d) is also the throughput results using the mixed workload with a 1:3 write and read ratio. We have similar observations as in the workload with a 3:1 write and read ratio, but Figure 12(d) shows higher throughput than Figure 12(c) because there are more reads than writes in the workloads.

4) REALISTIC WORKLOAD RESULTS

Figure 13 shows the results of throughput measurement for HACC-IO [26] using a variable number of MPI processes. Since there is little performance difference between array-based and tree-based SRL, we ran experiments with NOVA-SRL(tree). Figure 13 shows a similar trend to the DWOM workload evaluation results. NOVA-default exhibits lower scalability compared to NOVA with range-based locking due to the shared file writer-writer problem. The



(a) DWOM (write only)

(b) DRBM (read only)

(c) Mixed workload (75% writes, 25% reads)

(d) Mixed workload (25% writes, 75% reads)

FIGURE 12. Evaluation of the scalability of the NOVA file system according to various workload patterns on the Intel Optane DC server. Error bars are shown as average standard deviation.

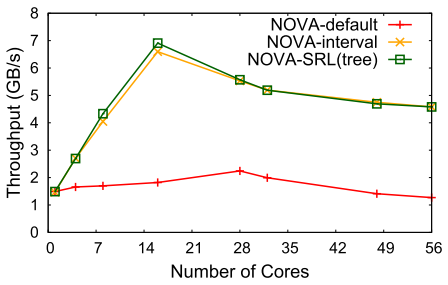


FIGURE 13. Throughput comparision for HACC-IO N-to-1 checkpoint workloads.

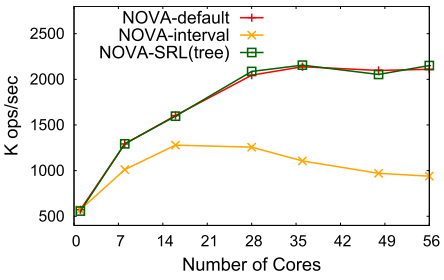


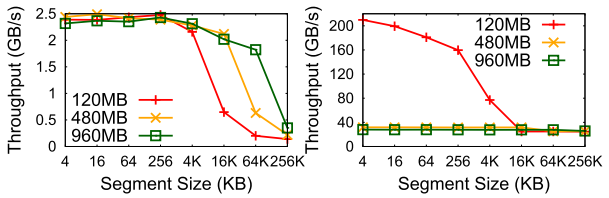
FIGURE 14. Throughput comparison with RocksDB by varying threads in DBbench workloads.

I/O throughput of NOVA-interval and NOVA-SRL(tree) increases as the number of cores increases, reaching up to 6.5 GB/s and 6.9 GB/s on 16 cores respectively. As a result, NOVA-SRL(tree) throughput is up to 3.8× higher than NOVA-default for the same reasons as in the DWOM workload.

Figure 14 shows the throughput comparisons for RocksDB by varying the number of application threads. The data read:write ratio for this workload is 9:1, but the result is completely different from the FxMark workloads experiment. NOVA-default and NOVA-SRL (tree) scale almost equally with respect to the increased number of threads. The reason why NOVA-SRL(tree) has not improved over NOVA-default can be attributed to the I/O pattern of the RocksDB as follows: unlike the DRBM experiment results, the CPU cache miss rate is very high because of the random read, so the read I/O performance is bound on the speed of reading data from the PM device. Therefore, the shared file reader-reader problem of the existing NOVA is not observed. Moreover, since the shared file write pattern rarely exists, the I/O parallelism by SRL did not work for this workload. On the other hand, NOVA-interval had low overall performance scalability due to the lock overhead protecting the interval tree as described in the DRBM experiment results. Rather, the performance of NOVA-defalut on 56 cores was 2.28× higher than that of NOVA-interval.

5) SENSITIVITY ANALYSIS BY SEGMENT SIZE

We analyzed the throughput of NOVA-SRL(tree) with different segment sizes and file sizes as shown in Figure 15. We



(a) DWOM (56 writers) (b) Modified DRBM (56 readers)

FIGURE 15. Throughput of NOVA-SRL(tree) according to the file size and segment size when threads concurrently perform I/Os on a shared file.

varied the segment size from 4 KB to 256 KB, and used three file sizes such as 120 MB, 480 MB, and 960 MB for the experiments. For the evaluation, we ran 56 I/O threads that performed the DWOM and DRBM workloads over the maximum number of CPU cores (i.e., 56 cores) in the Testbed. However, in the DWOM and DRBM workloads, it is difficult to measure sensitivity according to the segment size because threads repeatedly perform I/Os in each range. Therefore, we slightly modified FxMark so that it allowed threads to randomly access pages of a file where they access mutually exclusive areas between threads.

Figure 15(a) shows the average write throughput measured by varying file size and segment size. As expected, when the segment size increases, thread contention occurs in the *slv*, resulting in performance degradation. Whereas if the file size gets larger, thread contention for the *slv* is alleviated.

Figure 15(b) shows the average read throughput measured by varying file size and segment size. Unlike the DRBM workload, which reads the same data repeatedly, the modified random read pattern incurs last-level cache misses. In particular, as the file size increases, the spatial locality of threads decreases. To confirm this, we measured the cache miss ratios by increasing the file size. As expected, the cache miss ratio in a 120 MB file was only 2.61%, while the cache miss ratios in 480 MB and 960 MB files significantly increased to 33.85% and 55.53%, respectively. It is evident that in the cases of 480 MB and 960 MB file sizes, the cache is not sufficiently utilized. For the 120 MB file size, performance is gradually degraded as the segment size increases. This can be explained by the fact that the probability of competing for the read counter of *slv* increases as we increase the segment size, which results in increasing the cache line invalidation overhead.

VII. RELATED WORK

Range-based locking can selectively refine a critical section protected by a lock to increase the parallelism of a system. It is mainly used in a distributed file system where large data or metadata files are shared by many threads. GPFS [21] and Lustre [30], [31] are famous parallel distributed file systems that perform range locks for file synchronization. GPFS uses a negotiation protocol when nodes in a cluster start writing to a shared file, requesting a file area that other nodes are no longer using. Lustre also confirms that there is no conflict with the range of other I/O requests. Also, W. Liao [32] proposed a few file domain partitioning methods designed to reduce lock conflicts under locking protocol adopted by GPFS and Lustre. That is, they all perform a range comparison between lock competitors. In such a distributed environment, the overhead of range comparison may not be seen due to networking costs. However, considering the latest manycore server, the situation where multiple threads perform I/Os concurrently for large files is no longer an issue that occurs only in distributed environments. Therefore, in this paper, we analyzed the overhead of the existing interval tree-based range lock and proposed a segment-based range lock. Kogan *et al.* [33] proposed a linked list-based range lock and referred to our previous work [18]. According to [18], they claimed that the lower the degree of overlapping of the range, the higher the performance of the SRL. In a file system, it is generally rare to write and read while overlapping the same range in shared file I/O, so SRL is efficient.

Range-based synchronization has been performed in an attempt to design fine-grained locks. On the other hand, it is well known that the use of a non-scalable lock becomes a bottleneck of system performance scalability [19], [34]. In order to design scalable locks, researchers mainly reduce the number of accesses to lock variables competitively trying to acquire locks. This is because it brings congestion to the memory bus and causes big problems in performance [35]. In particular, as most large systems are manycore servers that support NUMA architecture, a new challenge has been given to the existing scalable lock design. Therefore, there have been many research efforts to design a NUMA-aware scalable lock even in a high-contended situation [22], [36], [37]. Cohort lock [22] and CST lock [36] use NUMA-aware hierarchical locking. They use scheduling techniques that give priority to lock competitors in the same socket, reducing the situation of sharing a cache line containing lock variables between different sockets. We can apply the mechanisms of these studies to our range locking in the future to address problems that have not yet been solved.

VIII. CONCLUDING REMARKS

In this paper, we implemented a variant of the NOVA file system with a fine-grained range-based RW locking method that locks only a part of the file, thus allowing parallel I/Os in a single shared file. In particular, we proposed a segment-based range lock called SRL suitable for the NVM file system in a manycore environment. We conducted various experiments on a manycore server equipped with real PM modules and evaluated the proposed approach using benchmarks in which multiple threads perform I/Os to the shared file area. The benchmarking results showed that the SRL-based NOVA outperforms the original NOVA by 3× in terms of write throughput. In addition, we found that the original NOVA did not utilize the cache efficiently, whereas our implementation fully utilized the cache. As a result, the read throughput scaled linearly as we increase the number of cores. From our investigation of various Linux file system architectures,

we expect that SRL can be easily employed in other file systems.

REFERENCES

[1] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 465–479, 2008.

[2] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush, "A 16 Gb ReRAM with 200 MB/s write and 1GB/s read in 27 nm technology," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 338–339.

[3] Intel. (2017). *Revolutionizing Memory and Storage*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[4] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 323–338.

[5] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst. (EuroSys)*, 2014, pp. 15:1–15:15.

[6] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. 9th Eur. Conf. Comput. Syst. (EuroSys)*, 2014, pp. 14:1–14:14.

[7] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst. (EuroSys)*, 2016, pp. 12:1–12:16.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ. (SOSP)*, 2009, pp. 133–146.

[9] S. Zheng, L. Huang, H. Liu, L. Wu, and J. Zha, "HMVFS: A hybrid memory versioning file system," in *Proc. 32nd Symp. Mass Storage Syst. Technol. (MSST)*, 2016, pp. 1–14.

[10] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2011, pp. 39:1–39:11.

[11] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proc. 26th Symp. Operating Syst. Princ. (SOSP)*, 2017, pp. 460–477.

[12] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the ZoFs user-space NVM file system," in *Proc. 27th ACM Symp. Operating Syst. Princ. (SOSP)*, 2019, pp. 478–493.

[13] R. Yujie, M. Changwoo, and K. Sudarsun, "CrossFS: A cross-layered direct-access file system," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 137–154.

[14] 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 *Proc. 26th Symp. Operating Syst. Princ. (SOSP)*, Oct. 2017, pp. 69–86.

[15] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf. (ATC)*, 2016, pp. 71–85.

[16] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Apr. 2019, pp. 427–439.

[17] J. Corbet. (Jun. 2017). *Range Reader/Writer Locks for the Kernel*. [Online]. Available: https://lwn.net/Articles/724502/

[18] J.-H. Kim, J. Kim, H. Kang, C.-G. Lee, S. Park, and Y. Kim, "pNOVA: Optimizing shared file I/O operations of NVM file system on manycore servers," in *Proc. 10th ACM SIGOPS Asia–Pacific Workshop Syst.*, 2019, pp. 1–7.

[19] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proc. 9th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, vol. 10, no. 13, pp. 86–93, 2010.

[20] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive reader-writer locks," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2014, pp. 219–230.

[21] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, vol. 2, no. 19, p. 19-es, 2002.

[22] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 247–256, 2012.

[23] *Intel*. Accessed: 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

[24] Intel. *Intel Memory Latency Checker V3.8*. Accessed: 2020. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[25] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 169–182.

[26] V. Vishwanath. *HACC I/O*. Accessed: 2012. [Online]. Available: https://github.com/glennklockwood/hacc-io

[27] *RocksDB*. Accessed: 2020. [Online]. Available: https://rocksdb.org/

[28] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W.-K. Liao, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astron.*, vol. 42, pp. 49–65, Jan. 2016.

[29] P. Stenström, T. Joe, and A. Gupta, "Comparative performance evaluation of cache-coherent NUMA and COMA architectures," in *Proc. 19th Annu. Int. Symp. Comput. Archit.*, 1992, pp. 80–91.

[30] P. J. Braam and P. Schwan, "Lustre: The intergalactic file system," in *Proc. Ottawa Linux Symp.*, 2002, vol. 8, no. 11, pp. 3429–3441.

[31] P. Braam, "The lustre storage architecture," 2019, *arXiv:1903.01955*. [Online]. Available: http://arxiv.org/abs/1903.01955

[32] W.-K. Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 260–272, Feb. 2011.

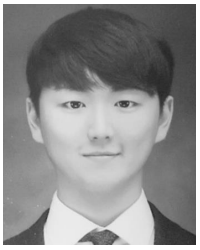
[33] A. Kogan, D. Dice, and S. Issa, "Scalable range locks for scalable address spaces and beyond," in *Proc. 15th Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA, 2020, pp. 1–15, doi: 10.1145/3342195.3387533.

[34] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proc. Linux Symp.*, 2012, pp. 119–130.

[35] V. Luchangco, D. Nussbaum, and N. Shavit, "A hierarchical CLH queue lock," in *Proc. Eur. Conf. Parallel Process.* Berlin, Germany: Springer, 2006, pp. 801–810.

[36] S. Kashyap, C. Min, and T. Kim, "Scalable Numa-aware blocking synchronization primitives," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2017, pp. 603–615.

[37] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, "NUMA-aware reader-writer locks," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 157–166.

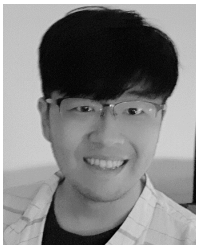


JUNE-HYUNG KIM received the B.S. degree from the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea, where he is currently pursuing the M.S. degree in computer science. He is a member of the Distributed Computing and Operating Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include operating systems, file and storage systems, and parallel and distributed systems.

YOUNGJAE KIM (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2001, the M.S. degree in computer science from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. He was a Research and Development Staff Scientist with the U.S. Department of Energy's Oak Ridge National Laboratory from 2009 to 2015 and as an Assistant Professor with Ajou University, Suwon, South Korea, from 2015 to 2016. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sogang University. His research interests include operating systems, file and storage systems, parallel and distributed systems, computer systems security, and performance evaluation.

SAFDAR JAMIL received the B.E. degree in computer systems engineering from the Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing the M.S. degree leading to Ph.D. degree in integrated program with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea.

He is also a member with the Distributed Computing and Operating Systems Laboratory, and with the Department of Computer Science and Engineering, Sogang University. His research interests include scalable indexing data structures and algorithms and memory-centric computing.



CHANG-GYU LEE received the B.S. degree in computer science from Ajou University, Suwon, South Korea. He is currently pursuing the M.S. degree leading to Ph.D. degree in integrated program with the Department of Computer Science and Engineering, Sogang University, Seoul. He is also a member with the Distributed Computing and Operating Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include operating systems, file and storage systems, key-value stores, and parallel and distributed systems.

SUNGYONG PARK (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, and the M.S. and Ph.D. degrees in computer science from Syracuse University, Syracuse, NY. He is currently a Professor with the Department of Computer Science and Engineering, Sogang University. From 1987 to 1992, he worked for LG Electronics, South Korea, as a Research Engineer. From 1998 to 1999, he was a Research Scientist with Bellcore, where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high-performance I/O and storage systems, and embedded system software.