pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers

June-Hyung Kim, Jangwoong Kim* Hyeongu Kang, Chang-Gyu Lee
Sungyong Park, Youngjae Kim
Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
{junehyung,ski6812,hyeongu,changgyu,parksy,youkim}@sogang.ac.kr

ABSTRACT

NOVA is a state-of-the-art non-volatile memory file system that logs on a per-file basis to ensure consistency. However, NOVA does not show scalability when multiple threads perform I/Os to a single shared file on Manycore servers. We identified two problems: First, when multiple threads write to a single file restricts parallel writes because of a coarse-grained lock on files in the file system layer. Second, when multiple threads read to a single file, every reader lock acquisition invalidates cachelines of waiting threads and block holders. In order to solve the aforementioned problems, we propose *pNOVA*, a variant of NOVA that accelerates parallel writes and reads to the same file of multiple threads. First, *pNOVA* employs a fine-grained range lock, for which we take two implementations, an interval tree based range locking and an atomic operation-based range locking, rather than a coarse-grained lock on files. Second, by defining a range locking variable per each file range, we alleviate the cacheline invalidation problem of a single read counter. Lastly, we address the potential consistency damage incurred by parallel writes to the shared file, and provide consistency using a commit mark based logging method. We evaluated *pNOVA* on a Manycore server with 120 cores. For microbenchmark, *pNOVA* showed up to 3.5× higher I/O throughput than NOVA for concurrent shared file write workload. In the Filebench-OLTP benchmark, *pNOVA* showed up to 1.66× higher transaction processing rate than NOVA.

CCS CONCEPTS

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

KEYWORDS

Non-Volatile Memory, File System, Operating System

APSys '19, August 19–20, 2019, Hangzhou, China

ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343748

1 INTRODUCTION

There have been several file system studies that use Non-Volatile Memories (NVM) such as phase change memory [7], resistive memory [3] and Intel and Micron's 3D-Xpoint [4] as a storage device [1, 2, 6, 9–12]. These NVM-based file systems provide higher throughput as well as lower read-write latencies than block device-based file systems. Among the most recent NVM-based file systems, NOVA [11] is one of the most advanced persistent NVM file systems, ensuring consistency of data and metadata in the event of a sudden power outage. To ensure this consistency, NOVA [11] implements a per-inode logging which logs meta-data for every write operation to each file. This per-inode logging provides not only high concurrency for handling transactions but also recovery process after crash, since the log of each file can be scanned simultaneously. In addition, NOVA has per-core data structures including NVM page allocator, inode table and journaling space. The concurrent nature of NOVA along with high performance NVM-based storage is expected to boost the performance of parallel I/O.

However, we identify that NOVA does not provide any degree of scalability in terms of I/O throughputs when concurrent I/Os are performed on a shared file on Manycore servers. This is mainly because of a coarse-grained lock per file in file systems [5], which negates any benefit of concurrent nature of NOVA and high performance NVM storage. Specifically, in NOVA, when a thread writes to a file, all threads attempting to write to the same file must wait until the previous thread finishes its operation. We also address that the single read counter variable of the read-write lock implementation degrades the performance of parallel read operations. In NOVA, the read thread begins by incrementing the reader lock variable. Since the lock is a shared variable, the increment becomes a serialization point and also invalidates the cachelines of other read threads and lock holders.

To solve this coarse-grained lock problem, we suggest a range-based reader-writer synchronization mechanism,

*He is currently affiliated with TmaxSoft.
Corresponding authors: Sungyong Park and Youngjae Kim

which selectively blocks I/O operations only with overlapped ranges with lock holders, and implements it in the NOVA file system, which is called parallel NOVA (*pNOVA*). In *pNOVA* with the range-based synchronization, there can be multiple writers at a given time, thus achieving write concurrency. In terms of read, we alleviate the cacheline invalidation overhead due to a single read counter by deploying multiple range locks. To effectively find overlapped ranges, we attempted to apply for an interval tree based range lock in NOVA. The interval tree is a well known data structure for detecting overlapped ranges. However, the range checking operation in the interval tree begins by holding the mutex lock of the tree and then performs tree insertion and traversal. We find that the coarse-grained lock of the interval tree becomes bottleneck in range checking process on Manycore servers. Next, we suggest an atomic operation-based range lock that performs better than an interval tree-based range lock. The atomic operation-based range lock uses hardware-supported atomic operations that ensure memory ordering.

When NOVA adopts the aforementioned range-based synchronization and multiple writers perform writes in parallel, the file system consistency might be broken. This is contributed by NOVA's log-structured logging for consistency: multiple writers have to write on the log simultaneously, competing for a single log pointer, which can damage consistency. We guarantee the file system consistency by adopting commit mark. At the end of each write operation, *pNOVA* commits the log entry by putting commit mark on the entry along with memory fence and cacheline flush instructions. Then in recovery phase, only entries with commit mark become recovery candidates. This commit mark-based logging safely completes the transaction with simultaneous multiple writers and makes the file system consistent after power failure.

For evaluations, we measure the performance of *pNOVA* using a mix of micro and realistic benchmarks, FxMark [5] and Filebench-OLTP [8] respectively. The experimental results show that *pNOVA* provide 3.5× and 1.66× higher throughputs for micro and macro benchmark respectively.

2 BACKGROUND AND MOTIVATION

NOVA is a hybrid memory file system which uses both persistent memory and DRAM. The consistency is guaranteed by using per-inode logging in persistent memory. The log records every modification of that file or directory. For user data, NOVA uses copy-on-write. Specifically, it first writes the new user data page(s) and records the log which is led to point the start page of user data later. Finally, the tail pointer is updated to the new log. Logs are contained in a 4KB log page in persistent memory. Both of log pages and user data pages are allocated by a per-cpu memory page allocator.

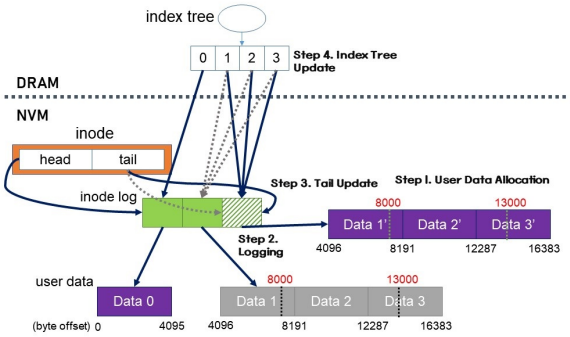

Figure 1: Write operation of NOVA [11]

The NOVA File System: Figure 1 illustrates the write operation of NOVA. For a write operation, NOVA first allocates data page(s) required to handle the write operation. Since the page is 4KB, enough number of pages need to be allocated to accommodate the range of the write operation. Then, the data in the previous pages that do not belong to the range of the write are copied to the new pages, followed by user data copy from the user buffer to the new pages (step 1). After the data pages are completely written, corresponding log entry is written which contains the information about the operation. The position where the log entry is written is decided by the tail pointer. The log is written right after the tail (step 2). However, if the tail points the last log entry in the log page, a new log page is allocated where the log entry is inserted in the head of the page. The tail is updated to the new log entry (step 3). Then, the index tree in DRAM is updated such that the new index node points to the new log entry (step 4), which finalizes the atomic write.

At the sudden power failure, NOVA can recover the file to a consistent status by scanning the inode's log. Beginning at the head pointer of the inode, all of log entries are scanned until it encounters the last log entry which the tail points to. This scanning process reconstructs the file index tree correctly. The write operation is atomic in a sense that the operation is executed completely or not executed at all at the occurrence of a power failure. If a crash occurs after the user data is written but before writing the log entry, the operation is not visible because no log entry points to the user data. In case of system crash after writing the log entry but before the tail update, the operation is also not visible, since the log is located after the tail, which makes the log excluded in the scanning process. The last case is when a crash occurs after the tail update but before the index tree update. Since the log entry is pointed by the tail, the scanning process includes that log entry, followed by index tree update to that log entry. Therefore, the user can access the newest data from the index tree, which means the write is performed completely.

Shared File I/O performance of NOVA: Although NOVA is a well-optimized file system for NVM with scalable data

structures and strong consistency guarantee, its per-file coarse-grained locks degrade the I/O performance in a shared file in Manycore server environments. We describe how the coarse-grained locks damage the performance of write and read operation in a single shared file on Manycore servers

In the write operation in a single shared file with multiple write threads, only one of I/O threads acquires the write lock of the file by calling *down_write*. The write operation is performed with the write lock being held, during which other write threads are blocked waiting for the lock to be released. Then, when the lock is released by the lock holder, one of the waiting write threads succeeds in acquiring the lock and performs the write. The coarse-grained write lock only allows a single writer to perform write, which prevents any concurrency among writers.

For the read operation, read threads also begin by holding the read lock using *down_read* function. Unlike the write lock, the read lock can be held by multiple readers, which means multiple read operations can be performed in parallel. However, the single read lock variable can be the performance bottleneck in Manycore environments. When a read thread acquires the read lock by calling *down_read*, the read lock variable is incremented by one. This increment invalidates the cachelines that correspond to the read lock variable of other waiting threads and also lock holders. In Manycore servers over a hundred of cores, the overhead of this cacheline invalidation can be critical.

3 DESIGN AND IMPLEMENTATION

3.1 Interval Tree Based Range Locking

Interval Tree: Interval tree is a well-known data structure designed to find all intervals overlapping with any given interval, which is also shipped in Linux kernel. Figure 2 illustrates interval tree. Each interval is represented as a node in a tree. A node consists of *start* and *end* of the interval. We can use any kind of ordered tree structures including binary search tree or red-black tree. A node is inserted with its *start* value as a key for sorting. While being inserted, it executes in-order traversal to find all overlapping intervals. We can skip this brute-force traversal by adding an extra value representing the maximum *end* value of all intervals in its child nodes, and this value will be denoted by *max* in this paper.

With *max* value in each tree node, when traversing the tree to find overlapping intervals, we can skip when satisfying one of the following conditions:

- all nodes to the right of nodes whose *start* value is bigger than the *end* value of the given interval
- all nodes that have their *max* values smaller than the *start* value of the given interval

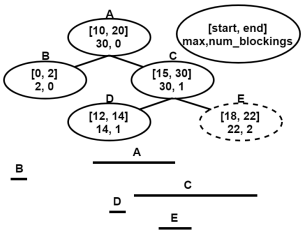

Figure 2: Interval tree

By the characteristic of an interval tree above, the time complexity of finding an overlapping interval can be *O(1)* with *n* as the number of intervals in the best case.

Interval Tree Based Range Locking: Range locking is a synchronization mechanism which locks only a part of an object. There exists a range lock implementation based on an interval tree in the form of Linux kernel patch. In the range locking mechanism, each range lock is represented as an interval tree node. And there is a single mutex lock to protect the interval tree.

For example, in the Figure 2, there are two range lock holders (with *num_blockings* zero) and two range lock waiters (with non-zero *num_blockings*), and a range lock (E) with range [22, 2] 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. The traversal can be shortened by skipping some eligible nodes.

Then, E stores the number of overlapping nodes to *num_blockings*. If *num_blockings* is zero, since it means 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 are 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.

To allow parallel I/Os on a single file, we can simply try applying the range locking to a file system. We used the interval tree based range lock primitive for our *pNOVA* implementation. An interval tree is defined for each file and it resides in DRAM, since the locking status does not need to be stored persistently. At the beginning of the file system's I/O function, *pNOVA* generates an interval tree node using start and end page numbers of the write request as its *start* and *end* value of the node. Then, *pNOVA* traverses the tree to find intervals that overlap with the given interval. During the traversal, *pNOVA* stores the number of overlapping intervals (which will be denoted by *num_blockings*) to its node. After the traversal, *pNOVA* inserts the node to the tree by using the *start* value as the key of sort.

However, we need to be aware that lock and unlock operations for a range lock are done while holding a mutex of a interval tree. The coarse-grained mutex of an interval tree can seriously damage the parallelism in Manycore server

Algorithm 1 Atomic Operation-based Range Write Lock

```
1:  void nova_atomic_write_lock (inode, start, end) {
2:      atomic_t *rwlock = inode.rwlock;
3:      unsigned int wlock = 1 << 31;
      /* 100000....0, when write lock is held */

4:      for ( cur=start ; cur <= end ; cur++ )
5:          while true :
6:              smp_mb_before_atomic()  // barrier
7:              old=atomic_cmpxchg(&bitmap[cur], 0, wlock);
8:              smp_mb_after_atomic()  // barrier
9:              if old_rwlock == 0:
10:                 break  // write lock succeeds only when 000000....0
      // namely, there is neither writer nor reader
11:     return;
```

Algorithm 2 Atomic Operation-based Range Read Lock

```
1:  void nova_atomic_read_lock (inode, start, end) {
2:      atomic_t *rwlock = inode.rwlock;
3:      unsigned int wlock = 1 << 31;
      /* 100000....0 */

4:      for ( cur=start ; cur <= end ; cur++ )
5:          while true :
6:              smp_mb_before_atomic()  // barrier
7:              old=atomic_add_unless(&bitmap[cur], 1, wlock);
8:              smp_mb_after_atomic()  // barrier
9:              if old_rwlock != 0:
10:                 break  // read lock always succeeds but when
      // 100000....0. namely, when there is a writer
11:     return;
```

environments, since the mutex lock serializes all incoming I/O requests. In particular, in NVM file systems with fast storage devices, the overhead can be more critical.

3.2 Range Locking using Atomic Operation

We propose a hardware atomic operation-based range locking that is more efficient than the interval tree-based range lock. Specifically, we define *segment*, a contiguous set of pages of a file. The segment size is a multiple of the page size. For each segment of a file, a 32-bit variable is defined which is used for a reader-writer lock for the segment. The left-most bit is for a writer lock. If the segment is being written, the bit is one and else the bit is zero. The remaining 31 bit work as a reader counter, the value of which is the number of active readers for the segment.

Algorithm 1 and 2 describe the implementation of writer and reader lock functions. To atomically acquire the writer and reader lock, we used atomic operations supported by the hardware. In Algorithm 1, the writer tries to acquire a writer lock from *start* to *end* segments. The variable *wlock* is the binary representation of a reader-writer lock variable when the writer lock is held.

For each segment in [*start*, *end*], *pNOVA* tries to acquire the writer lock by calling *atomic_cmpxchg* function, which is a hardware atomic operation. The function changes the value of the reader-writer lock of the segment to *wlock* only when the value of lock is 0, and returns the old value of the lock. If the old value of the lock is 0, which means there is no writer or reader, the value of the lock is changed to *wlock*, and the writer lock succeeds. Otherwise, if the old value of the lock is non-zero, the value of the lock is unaffected, and the writer lock fails. Then, the writer threads tries to acquire the writer lock again.

We also use atomic operations for the reader lock. A read thread trie to hold a reader lock by increasing the reader counter by calling *atomic_add_unless* function which is also a hardware atomic operation. The function increments the value of the reader counter by 1 only when the value of the counter is not *wlock*, which means that the reader lock fails only when the writer lock is held, and succeeds otherwise.

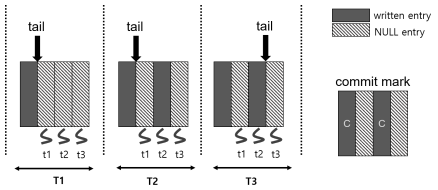

Figure 3: An example of inconsistent file system status after a power failure and a description of the Commit Mark data structure. $T1$, $T2$, and $T3$ threads represent the order of time intervals, and C represents commit mark.

This also implies that there can be multiple readers for the segment, since writer lock always succeeds unless the left-most bit is 1.

To unlock the writer and reader lock, it does not need to repeatedly try atomic operations. For writer unlock, we simply clear the left-most bit using *clear_bit* atomic operation. Note that since any writer or reader lock acquisitions cannot succeed until the writer lock holder releases the lock, it is always safe to clear the left-most bit. In reader unlock, we just decrease the reader counter by 1 using *atomic_dec*.

For the correctness of the range locking, we call *smp_mb_before_atomic* and *smp_mb_after_atomic* before and after of each atomic operation. They are memory barriers that prevent memory ordering for the atomic operation. The atomic operation-based range locking has less overhead than the interval tree-based range lock because it is not a coarse-grained locking, which was the major problem of the range lock based on an interval tree.

Analysis of Memory Space Overhead: Each segment requires a 32-bit variable. That is, the size of the segment and the maximum size of the file determine the additional memory space required for atomic operation-based range locks. For example, when the maximum file size is 1GB and the segment size is 4KB, 2^{18} segments are required and 1MB of memory space is required for lock variables.

3.3 Guaranteeing Consistency

The NOVA write operation updates the tail after appending the log entry to validate the corresponding log entry. Tail

Algorithm 3 Commiting Log Entry

```
1: new_tail = append_to_log(tail, entry)
2: clwb(tail) /* writes back the log entry cachelines */
3: sfence() // orders the subsequent store
4: entry− >committed = TRUE // commit the entry
```

update, however, might lead the file system to an inconsistent state when multiple threads perform shared file writes concurrently. Figure 3 illustrates an example of potential inconsisteny problem. Suppose in time interval $T1$, three threads ($t1$, $t2$, $t3$) are trying to perform write to a shared file. At this time, each thread is given one log entry. Figure 3 shows each thread is given one log entry. In $T2$, suppose the second thread ($t2$) writes data pages and writes the log entry, before the first thread ($t1$) and third thread ($t3$) write their log entries. Then, in $T3$ the tail is updated to point to the log entry of the second thread. Suppose a sudden power failure occurs. After the system is restarted, the file is recovered by scanning log entries until it encounters the last entry which the tail points to. Then, recovery will include the invalid log entry such as that of $t1$ in this example. This leads to an inconsistent file system status.

To solve this problem, *pNOVA* adopts commit mark to validate the log entry. Algorithm 3 describes the consistency guarantee using commit mark. After a log entry is appened, *pNOVA* adds a commit mark to the entry. Memory fence and cache line flush operations are also used to guarantee the memory order of the commit mark and log entry update. This is the same method used to update tail in the original NOVA. Therefore, commit mark allows to guarantee consistency with the same overhead as NOVA.

4 PERFORMANCE EVALUATION

Experimental Setup: We measure the parallel I/O performance of *pNOVA* and NOVA on a Manycore server with 120 cores. The detail specification of our testbed is described in Table 1. Three versions of NOVA were compared: baseline NOVA, pNOVA that uses an interval-tree based range lock (pNOVA-interval), and pNOVA that uses an atomic operation based range lock (pNOVA-atomic). The segment size is 4KB, which is the page size. We used a mix of synthetic and realistic benchmaraks (FxMark [5] and Filebench-OLTP). In particular, FxMark [5] is a file system scalability micro-benchmark. FxMark includes several patterns of workloads. Of these, we specifically use DWOM and DRBM workloads. The DWOM workload performs multi-threaded shared file writes and the DRBM workload performs multi-threaded shared file reads. In each workload, each thread continuously issues a write/read request to a mutually exclusive area of a file between threads.

Microbenchmark Results: Figure 4 and Figure 5 show the I/O throughput comparisons of NOVA and *pNOVA* for DWOM and DRBM workloads of FxMark. In DWOM, among

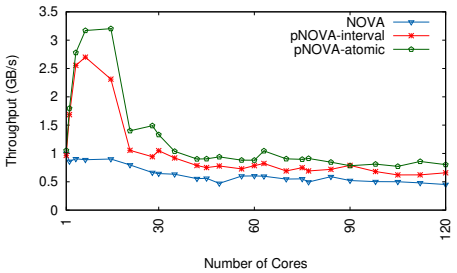

Figure 4: DWOM workload results in FxMark

the three, pNOVA-atmoic shows the highest throughput. However, its performance scalability is only guaranteed to a certain number of cores. Both pNOVA-interval and pNOVA-atomic scale up to 7 threads because the threads can perform writes with non-overlapping ranges in parallel. pNOVA-atomic shows 3.5X improvement compared to NOVA with 15 cores, which has higher improvement than pNOVA-interval. This is due to its light way implementation of acquiring the lock which is simply setting the corresponding bit in pNOVA-atomic. However, NOVA does not provide any scalability due to its coarse-grained lock.

In DRBM, NOVA provides some degree of scalability up to 15 cores. This is contributed by the use of *inode_lock_shared*, the reader lock of NOVA. Multiple readers can increment the reader counter while allowing read operations in parallel. However, since each lock acquisition modifies the value of a reader lock variable, this invalidates the cachelines for the lock varaible of the waiting readers and even lock holders. This cacheline invalidation overhead becomes more critical after 15 cores, which is the NUMA boundary of the testbed machine. Therefore, NOVA fails to scale after 15 cores.

pNOVA-interval does not provide any scalability. For every reader lock acquisition, the reader first holds the coarse-grained interval tree lock, inserts itself to the tree, traverses the tree to find overlapping nodes, and then releases the lock. The coarse-grained lock of the interval tree completely serializes the reader lock acquisition process and negates any benefit of multiple threads.

However, pNOVA-atomic shows great scalability up to 120 cores. Since the reader-writer lock varaible is defined for each segment of the file, the reader lock acquisition does not invalidate the cachline. Therefore, there does not exist any serialization point in acquiring the reader lock. In addition, in the DRBM workload, since each thread reads the same data

Table 1: Specification of the Server Machine

CPU	Intel(R) Xeon(R) CPU E7-8870 v2 2.30GHz CPU Node (#): 8 Cores per Node (#): 15
Memory	DDR3, 96 * 8 GB (=968GB)
NVM	32 GB (Emulated over DRAM)

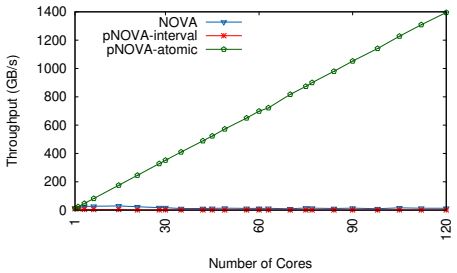

Figure 5: DRBM workload results in FxMark

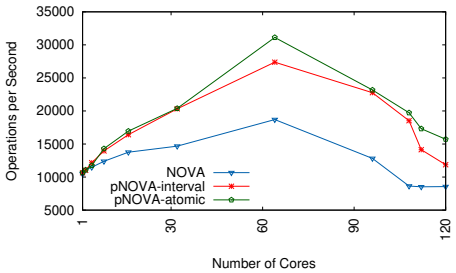

Figure 6: Scalability evaluation for Filebench-OLTP

repeatedly, cache misses for the data can barely occurs. This leads to greatly high throughput nearly up to 1,400 GB/s.

Macrobenchmark Results: We use Filebench-OLTP database benchmark. Table 2 summarizes the detail of the benchmark. We preliminarily ran benchmark with 64 threads and gathered all write I/O, and calculate *parallelizable write ratio*. We grouped every write request such that each group includes 8 sequential writes. Then, we say a group is *parallelizable* if *all 8 writes have exclusive ranges*, AND *all 8 writes are issued by different threads*. In Filebench-OLTP, threads issue 4 KB writes at a random offset. Then, it periodically issues 16 KB bulk writes to the database file, followed by *fsync*. The *fsync* system call of NOVA is, however, a NULL function, because NOVA ensures that once a write operation is performed, it is persistently stored on the storage medium immediately. Therefore, the overhead of fsync is not counted.

Figure 6 shows the performance comparisons of NOVA and *pNOVA* for Filebench-OLTP. Unlike DWOM, NOVA also shows scalability up to 60 cores. In DWOM, threads only request write system calls to a single file, and a single file write takes an absolutely large portion of the program's execution time. However, since Filebench-OLTP is designed to mimic database systems, I/O is not as frequent as DWOM. Therefore, single file writes do not take an absolutely large portion of the execution time of the program, making NOVA also show some degree of scalability.

Table 2: Specification of Benchmarks

	filebench-OLTP
size of DB file	100 MB
# of DB files	1
size of buffer pool	N/A
# of I/O threads	8
write I/O size	4 KB
parallelizable write ratio	69.04%

Table 3: CPU Utilization in Filebench-OLTP

	range check	spinlock	logging	etc
NOVA	98.42%	0%	0.2%	1.38%
pNOVA-interval	93.94%	3.88%	0.58%	1.6%
pNOVA-atomic	1.83%	95.06%	1.33%	1.78%

Both pNOVA-interval and pNOVA-atomic provide improvements compared to the original NOVA. In 64 cores, pNOVA-atomic provides 1.6X improvement, which is slightly higher than pNOVA-interval. As shown in Table 2, *parallelizable write ratio* of filebench-OLTP is 69.04%, so concurrent writes of *pNOVA* contributes to the improvement of performance.

Table 3 shows the CPU usage by functions in Filebench-OLTP with 64 threads. The ratio of each function is calculated by taking the CPU usage of NOVA's write function as 100%. Spinlock is used for *pNOVA* to protect ciritical sections in the write function, including atime update and log pointer increment. Original NOVA does not need spinlock because only one thread writes to a file. For every case, range check and spinlock accounted for more than 95% of CPU usage. Original NOVA spends more than 98% CPU time in coarse-grained file lock. Interval tree lock spends about 94% of CPU time to check the overlapping ranges. This accouts for large portion because of the contetion to the interval tree. Atomic lock spends less than 2% of CPU time to hold range lock due to its simple way of range checking. However, spinlocks account for about 95% of CPU time to protect critical sections. CPU time spent by logging and others, which are the actual parts of the write process, is 1.58%, 2.18% and 3.11% for NOVA, pNOVA-interval and atomic respectively.

5 CONCLUSION

We have addressed that coarse-grained lock incurs performance bottleneck in shared I/O of NOVA file system. Therefore, we have proposed *pNOVA*, a variant of NOVA file system with range reader-writer locking method that locks only a part of the file, thus allowing parallel I/O in a single shared file. To implement *pNOVA*, we used two range checking mechanisms. We have adopted interval tree, a well-known data structure for checking overalpping ranges. We also have proposed a new range checking machanism built with hardware atomic operations, which is more parallelizable than interval tree. Experimental results including micro and macro benchmarks have shown that *pNOVA* using atomic operations outperforms the original NOVA, while guaranteeing file system consistency.

ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2017R1D1A1B03032763).

REFERENCES

[1] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 133–146.

[2] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 15:1–15:15.

[3] Richard Fackenthal, Makoto Kitagawa, Wataru Otsuka, Kirk Prall, Duane Mills, Keiichi Tsutsui, Jahanshir Javanifard, Kerry Tedrow, Tomohito Tsushima, Yoshiyuki Shibahara, et al. 2014. 19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Proceedings of the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 338–339.

[4] Intel. 2017. Revolutionizing Memory and Storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

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

[6] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. 12:1–12:16.

[7] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.

[8] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *;login: The USENIX Magazine* 41, 1 (March 2016), 6–12.

[9] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 14:1–14:14.

[10] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 39:1–39:11.

[11] 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.

[12] Shengan Zheng, Linpeng Huang, Hao Liu, Linzhu Wu, and Jin Zha. 2016. HMVFS: A Hybrid Memory Versioning File System. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14.