2018 IEEE 36th International Conference on Computer Design

A Highly Non-Volatile Memory Scalable and Efficient File System

Fan Yang*‡, Junbin Kang†, Shuai Ma*‡, Jinpeng Huai*‡
*SKLSDE Lab, Beihang University, China †Tencent, China
‡Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China
{yangfan16, kangjb}@act.buaa.edu.cn {mashuai, huaijp}@buaa.edu.cn

***Abstract*—With the rapid development of fast and byte-addressable non-volatile memories (NVMs), hybrid NVM/DRAM storage systems become promising for computer systems. Existing NVM file systems have already been optimized around the NVM properties. However, they inherit some design choices of block-oriented storage devices that lead to scalability bottlenecks and data copy overhead for ensuring data consistency.**

In this paper, we present noseFS, a highly non-volatile memory scalable and efficient File System. It is designed to achieve high performance through a bundle of novel techniques: (1) a scalable lightweight naming integrating VFS with the underlying file system namespace, (2) a fine-grained byte-unit file index tree avoiding redundant copy overhead introduced by Copy-On-Write, (3) a lightweight journaling providing atomicity and scalability on many-core platforms, and (4) a lightweight *atomic-mmap* providing strong consistency guarantee with low overhead by tracking dirty pages. Experimental results show that noseFS performs much better than the state-of-the-art file systems with equally strong data consistency guarantees, and achieves near-linear scalability on a 40-core machine.

I. Introduction

Emerging non-volatile memories (NVMs) such as Phase Change Memory (PCM) [1] and Spin-Transfer Torque Memory (STT-RAM) [2] provide high performance comparable with DRAM and support fast and byte-addressable accesses through memory bus. The advent of the fast non-volatile memory technologies is expected to completely revolutionize the landscape of existing storage systems, and to deliver extremely high parallelism and low-latency data accesses.

Designing a highly scalable, efficient and consistent file system for emerging NVMs raises several challenges that need to be solved by fully exploiting NVMs characteristics such as byte-addressability. As the processor supports only 64-bit atomic writes, existing mainstream file systems use shadow paging, log-structured or journaling techniques to provide consistency guarantee, which requires ordered memory writes [3]. However, modern processors may reorder store operations to improve performance. Although the x86 architecture provides cacheline flush (CLFLUSH) and memory fence (SFENCE) instructions to enforce write ordering, flushing the CPU cache-line incurs a substantial performance overhead [4]–[6].

To overcome the above issues and to exploit the high performance offered by NVMs, existing NVM file systems (NVMFSs) [5]–[9] enhance the traditional disk-based file system designs by employing NVMs characteristics, and reduce the number of *CLFLUSH* by applying improved consistency techniques. However, they still retain the following drawbacks.

First, VFS, as the abstraction layer of underlying file systems, becomes cumbersome on NVMs when performing directory operations (i.e., metadata intensive operations) [6]. VFS supports generic semantics through in-memory objects, hierarchical naming, and synchronization. Specifically, in-memory *inodes* and *dentries* need to be allocated, initialized, and destroyed; the hierarchical naming resolves each pathname component; synchronization supports concurrent operations by using locks. However, supporting generic semantics causes performance bottlenecks [9], and the global *rename_lock* used in synchronization leads to scalability bottlenecks [10].

With the development of fast and byte-addressable NVMs, the in-memory caches in VFS (including the in-memory objects, and the *dentry_hashtable* and the *inode_hashtable* in hierarchical namespace) are no longer necessary, as they are introduced due to the slow data access of traditional block-based storage devices. In fact, the underlying file systems on NVMs are able to organize their directory structures in a fine granularity and support fast lookups.

Second, supporting data consistency guarantee is a performance killer for common file system operations. It comes at the price of the write amplification when using the journaling technique to guarantee data consistency. Moreover, the copy overhead introduced by adopting Copy-On-Write (COW) for file data updates and providing consistency for memory-mapped files is heavy for file operations.

Existing journal-based NVMFSs [6], [7] adopt COW for file data updates, and log only metadata to avoid writing data twice. However, the journaling overhead increases proportionally with the size of the metadata to be modified; COW requires copying the unmodified data in the overwritten block to the new allocated space (as the file data is organized in block granularity), which introduces heavy copy overhead when the application overwrites only a few bytes in each block.

The byte-addressable NVMs make the *DAX-mmap* come true, and applications can access NVMs directly by mapping the physical address in NVMs to the virtual address in DRAM. Unfortunately, the *DAX-mmap* cannot provide consistency guarantee for memory-mapped files when system crashes. The existing NVM-based file system NOVA [6] uses COW for

This work is supported in part by the National Key Research & Development Program of China 2016YFB1000503 and NSFC U1636210. For any correspondence, please refer to Shuai Ma.

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00071

memory-mapped file updates, and atomically copies the entire memory-mapped pages to data blocks in NVMs when synchronizing files, to provide consistency guarantee for memory-mapped files (which is called as *atomic-mmap*). However, it introduces heavy copy overhead for *msync* operation when an application modifies only a small portion of a memory-mapped region, as the unmodified memory-mapped pages that are read only do not need to be synchronized at all.

Contributions & Organizations To address the above issues, we propose a highly non-volatile memory scalable and efficient File System (noseFS).

(1) We design a lightweight naming to fully exploit the characteristics of NVMs by integrating conventional VFS with the namespace of underlying file systems (Sections III-B and C). The lightweight naming merges the double lookup, insertion and removal of file system objects launched by VFS and the underlying file system into a single one, and enables the careful crafting of DRAM data structures such as separating the global locks that cause scaling bottlenecks.

(2) We develop *Non-Copy-On-Write* by using the fine-grained byte-unit file index tree for the file structure to avoid redundant copy overhead introduced by Copy-On-Write (Sections III-B and D), and the lightweight journaling by logging only the address of the metadata to reduce logging overhead when providing consistency guarantee (Section III-B and E).

noseFS manages the *extent* in byte granularity instead of block granularity. When performing *overwrite* operation, it splits the associated *extent* into multiple ones, and writes only the updated data without copying unmodified data to the new allocated space. In addition, noseFS logs the metadata address only instead of the metadata content, reducing the overhead introduced by double metadata writing and cacheline flushing.

(3) We develop a lightweight *atomic-mmap* to achieve the data consistency guarantee with low overhead by tracking dirty pages (Section III-D). noseFS tracks the process of page fault interrupts, and records dirty pages. When synchronizing memory-mapped pages by *msync*, noseFS writes only the dirty pages to NVMs instead of the whole memory-mapped pages.

(4) We implement a highly scalable, efficient and consistent file system noseFS, and experimentally demonstrate that it outperforms the state-of-the-art NVM-based file systems NOVA [6] and PMFS [7] (Section IV).

II. Design Principles

Our work aims to design a file system that provides highly efficient, concurrent and atomic common system calls for user applications, and achieves high performance and scalability together with strong consistency guarantees on many-core systems by fully exploiting the NVMs characteristics. In this section, we discuss the design principles that guided this work.

(1) *A dedicated lightweight naming to support efficient and scalable directory operations on NVMs.*

We bypass the in-memory cache in VFS and redesign the file system namespace by fully exploiting NVMs characteristics, and propose the lightweight naming to offer hierarchical naming and synchronization like VFS. We unify *inode* and *dentry* into a single structure, and use the simple and efficient global hash table in NVMs to organize it, improving the performance of directory operations. The redesigned namespace enables the use of fine-grained locks (local locks) on each hash bucket instead of the global lock on the global hash table, supporting high concurrent directory operations.

(2) *A lightweight consistency mechanism to improve the performance of file system operations on NVMs.*

We present the lightweight journaling to provide data consistency for efficient file system operations, and propose the Non-Copy-On-Write (NCOW) and the lightweight *atomic-mmap*, to reduce copy overhead and support highly efficient file operations. The lightweight journaling logs only the metadata address instead of the metadata content in NVMs to further reduce the overhead introduced by the double metadata writing and the cacheline flushing. The NCOW is implemented by the fine-grained byte-unit file index tree to substitute the conventional block-grained organization (the size is typically 4KB or larger), and it aims to eliminate copy overhead when updating file data by COW. The lightweight *atomic-mmap* tracks the process of accessing memory-mapped pages, and records the dirty pages to synchronize only the modified pages instead of the whole memory-mapped pages, reducing the copy overhead when performing *msync* operation.

III. System Implementation

Based on the previous design principles, we now present the implementation details of noseFS, a highly scalable, efficient and consistent file system for NVMs. noseFS is implemented on Linux 4.6.5 kernel. It is not implemented as a Linux kernel module, as it bypasses the conventional VFS. Instead, it is combined with the Linux kernel by intercepting system calls to check whether the pathname is under noseFS, and, therefore, it performs its own operations to handle system calls.

A. Overview

In noseFS, all the file system metadata is stored in a single fixed-size structure called *pinode*, and *pinodes* are organized with a global hash table. This approach has three important advantages. First, since the two structures (i.e., *inode* and *dentry*) are unified into a single one, the in-NVMs data structures are simple and efficient. Second, since there is only one structure that needs to be persisted, the number of high latency *CLFLUSH* is reduced. Third, since the in-memory cache provided by VFS is bypassed, the highly concurrent path resolution, which is frequently used by directory operations such as *create, delete and rename*, becomes possible, and can be supported by the global hash table.

The file structure in noseFS is organized by the widely used extent-based B+ tree (or simply file index tree). As a file index tree can be rebuilt from its leaf nodes, noseFS builds it in DRAM, and only keeps its leaf nodes (*extents*) in NVMs, making the in-NVMs data structures simple and efficient.

Consistency in noseFS is enforced by using redo journaling, which means the updates are atomically logged, and made durable before being committed to the file system. To reduce the journaling overhead, noseFS uses Copy-On-Write for modified file data, and records to the journal only the metadata about the updates (i.e., *pinodes* and *extents*). To further reduce the overhead, noseFS records only the address of the metadata instead of the content. Specially, it first allocates new metadata with the latest contents, and keeps them still invalid before committing log entries atomically. Then noseFS records only the addresses of the latest versions metadata, and commits the updates to the file system only after finishing logging.

In addition, NCOW is proposed in noseFS to eliminate the copy overhead introduced by COW. noseFS adopts the fine-grained byte-unit file index tree, which consists of fine-grained byte-unit leaf nodes and internal nodes, to support NCOW.

B. Layout and Data Structures

The overall architecture of noseFS is shown in Figure 1. The layout consists of five parts: *superblock*, metadata structure, file structure, journal and space management.

Superblock The *superblock* contains the global file system information, e.g., a pointer to the root directory of a file system, and a pointer to the first log of a journal.

Metadata structure noseFS uses a dedicated array of 128-byte *pinodes*, referred to as the *pinode* table, to manage all *pinodes*, and it can easily locate a *pinode* given its number. Each *pinode* represents a file or directory in the file system. It contains (a) a 4-byte unique identification number (*pino*), (b) a valid bit, so that invalid ones can be reused for new files or directories, (c) file properties, including filename, (d) pointers to its file index tree, parent, children list, and file log list, and (e) pointers to the previous and next nodes in the associated lists (that is, hash table, directory and hard linking lists).

The global hash table in noseFS is an array of buckets, which has the same amount (128M) as the *dentry_hashtable* in VFS. Each bucket points to a list of *pinodes* with the same hash value, which is calculated with the corresponding file name and the *pino* of their parent directory.

noseFS keeps a mirror of the global hash table in DRAM for fast search. It adopts per-bucket locks to protect the updates of each bucket and further keeps the synchronization of the two hash tables in DRAM and NVMs. As will be seen in Section III-C, it uses per-bucket *seqlocks* to protect the updates of multiple buckets in an operation.

For a file with multiple filenames caused by hard links, which are created by *link* operations, each filename is stored in an individual *pinode* instead of *dentry*. All the *pinodes* of the file form a linked list, and each *pinode* contains a pointer to the first created *pinode*, which means the *pino* of the first created *pinode* will be provided when the *inode* number of a file is required. Only the first created *pinode* maintains the file data information, such as file size and the pointer to the file index tree, and the file and the first created *pinode* cannot be deleted unless there are no other hard links.

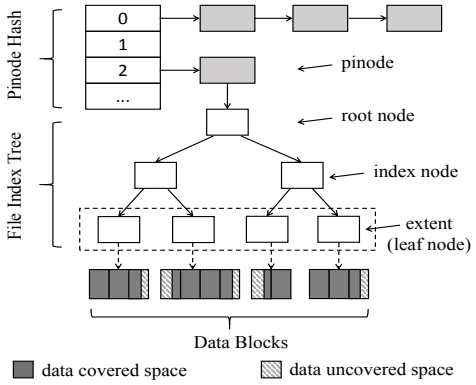


Fig. 1. Overall architecture of noseFS.

To alleviate the scalability bottleneck, noseFS uses a list of *pinodes* in DRAM for each CPU to manage the allocation and deallocation of *pinodes* in NVMs.

File structure noseFS uses a dedicated array of 40-byte *extents*, referred to as the *extent* table, to manage all *extents*, and it can easily locate an *extent* given its number. Each *extent*, as a leaf node of file index tree, represents a continuous region of file data in NVMs in byte granularity. It contains (a) a 4-byte unique identification number (*eno*), (b) a valid bit, so that invalid ones can be reused, (c) the start block number and length of the continuous 4KB data blocks, (d) the start offset and size of file data in the region in bytes, (e) a pointer to the *pinode* of its file, and (f) pointers to other *extents*.

The file index tree uses the start offset in *extents* as the search key, and each of its internal nodes holds at most 64 child nodes (Figure 1 only shows two for simplicity). noseFS stores only the *extents* in NVMs and organizes them with a linked list. The *pinode* of the file records the *eno* of the list head as the pointer to the file index tree. Each *extent* contains pointers to its neighbor *extents* in the list. As will be seen in Section III-D, the *write* operations are performed on the file index tree in DRAM only, and the synchronization of *extents* between DRAM and NVMs is executed by the *fsync* operation.

To alleviate the scalability bottleneck, noseFS uses a list of *extents* in DRAM for each CPU to manage the allocation and deallocation of *extents* in NVMs.

Journal noseFS provides directory and file journals separately for directory and file operations, and both journals use multiple logs to allow concurrent operations. It atomically records all the updates of an operation to a single log instead of multiple logs to avoid recovery ordering. The directory or file journal is a dedicated array of 512-byte logs, and each log is an array of 8-byte log entries. noseFS can easily locate a log or a log entry given its number.

All logs in the directory journal, as shown in Figure 2, evenly record the updates of *pinodes* in all hash buckets, as directory operations only involve the updates of the *pinodes*. The updates of *pinodes* in each hash bucket are only logged into a single corresponding log. For the updates of *pinodes* in multiple buckets in an operation, which may be logged into multiple corresponding logs, noseFS records the updates only

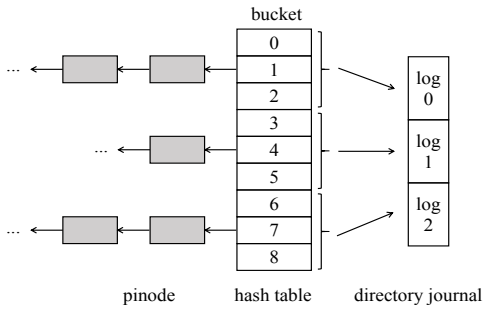


Fig. 2. Mapping of hash buckets to directory logs.

to the log with the largest log number, simultaneously holding multiple locks of logs sorted in ascending order of the log numbers, to guarantee the atomicity of the updates.

The first log entry of a log in the directory journal records the log tail, which stores the amount of the committed log entries. Each of the other log entries records the operation type and the *pino* of the updated or deleted *pinode*.

A log in the file journal records only the updates of *extents* of a single file. Although common file operations involve the updates of both the *extents* and the *pinode* of the file, each *extent* records a pointer to its corresponding *pinode* and the *pinode* can be updated according to the *extents*. Specifically, the file size recorded in the *pinode* can be calculated by the size recorded in *extents* during recovery, and the modified time is set as the recovery time. If the log of a *pinode* is full, noseFS extends it by building a list of logs, and the *pinode* records the pointer to the list.

The first log entry of a log in the file journal records the valid bit, which enables the reuse of the invalid ones, and the log tail, which stores the amount of the committed log entries. The last log entry records a pointer to the next log in the *pinode* log list. Each of the other log entries records the numbers of the updated *extents* of the old and latest versions, so that the updates can be performed according to the differences of the two versions of the *extent* during recovery. A log entry for a *delete* operation records the *eno* of the latest *extent* as *MAX*, to distinguish it from an update operation.

Space management noseFS manages the data space in NVMs using an array of 4KB data blocks, and uses the space-saving *bitmap* in NVMs to record the usage of data blocks (that is, each bit in the bitmap is corresponding to a data block).

To alleviate the allocation and deallocation contention of data blocks, noseFS evenly divides the data space into multiple regions, one per CPU, and uses a B+ tree in DRAM to organize the addresses of the free data blocks for each region. noseFS first allocates data blocks from the B+ tree of a current CPU. If no blocks are available, it then allocates blocks from the B+ trees of other CPUs. noseFS uses per-region locks to support concurrent operations.

The data blocks are deallocated to their allocated B+ trees (the original data region), and the leaf nodes are merged if the neighbor leaf nodes represent a continuous space, to increase the possibility of large space allocations.

C. Directory Operations

The lightweight naming in noseFS offers hierarchical naming and synchronization by organizing *pinodes* with the global hash table. We now explain how the lightweight naming in noseFS provides high performance and concurrency for the common directory operations.

When performing the path resolution, which is frequently performed by common directory operations, to find the target *pinode* of a file/directory, noseFS starts with the first component of the pathname, and resolves it to a known *pinode* (i.e., root or cwd). Then noseFS finds the child directory of that *pinode* according to the next component of the pathname, by traversing the corresponding hash bucket in DRAM. noseFS repeats the lookup of the child according to the next components until the previous one of the last component is resolved (that is, the corresponding *pinode* is found).

To support high concurrent operations, noseFS adopts per-bucket Read-Copy-Update (RCU) locks to protect the insertion and removal from each bucket of the global hash table and its mirror (like VFS does [11]). For the updates of two buckets caused by the *rename* operation, noseFS uses per-bucket *seqlocks* for synchronization, instead of using the global *seqlock* (*rename_lock*) like VFS. Specifically, the two buckets are locked simultaneously by the two *seqlocks* sorted in ascending order of the hash values to avoid deadlocks.

Creating a file In the *create* system call, noseFS first allocates a new *pinode* and inserts it into the mirror hash table in DRAM. Then it allocates and initializes a new *pinode* in NVMs, and logs the updates. After making the new *pinode* and the log entry durable, noseFS atomically updates the log tail to finish logging. Then it sets the valid bit of the new *pinode*, and inserts the *pinode* into the hash table in NVMs. Finally, it cleans the log by atomically updating the log tail.

Deleting a file In the *unlink* system call, noseFS first removes the *pinode* from the mirror hash table in DRAM. Then it logs the updates and makes the log entry durable. After atomically updating the log tail, noseFS sets the *pinode* invalid, and removes it from the hash table in NVMs. If there are no other *pinodes* pointing to the same file, noseFS frees the file index tree in DRAM and the *extents* in NVMs. Finally, it atomically updates the log tail to clean the log.

Renaming a file In the *rename* system call, noseFS first removes the *pinode* from its source bucket, and inserts the updated *pinode* into its target bucket of the mirror hash table in DRAM. Then noseFS allocates and initializes a new *pinode* in NVMs. As the updates of two buckets may be logged into two directory logs, the operation cannot be recovered atomically if a crash happens after updating the log tails. noseFS then encapsulates the two updates together as a transaction, and commits it only to the log with the larger log number. After making the new *pinode* and the log entries durable, noseFS atomically updates the log tail, and updates the source and target hash buckets in NVMs. Finally, it atomically updates the log tail to clean the log.

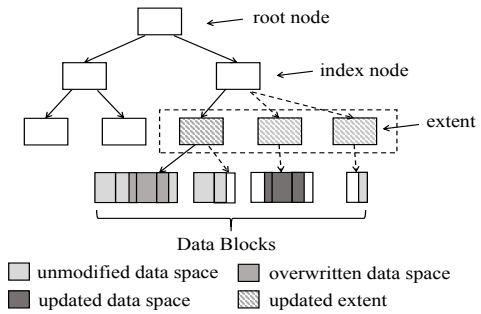


Fig. 3. The Non-Copy-On-Write mechanism.

D. File Operations

noseFS proposes Non-Copy-On-Write for updating file data to eliminate the copy overhead, and achieves it using the fine-grained byte-unit file index tree. For *append* operations, noseFS simply allocates new data blocks to write data, and inserts the corresponding *extents* into the file index tree. For *overwrite* operations, as shown in Figure 3, noseFS first searches the file index tree to find the original *extent* that manages the overwritten data. It then allocates new data blocks to write only the updated data, and inserts the corresponding *extents* into the file index tree. For the unmodified part of the data, noseFS splits the original *extent* into two new *extents* to manage them, instead of copying them to the new data blocks.

Writing to a file In the *write* system call, noseFS first searches the start *extent* in the file index tree with the write position, then uses *NCOW* to perform the updates. It should be pointed out that noseFS supports the *batch-write* operation, and does not synchronize the updates of *extents* to NVMs until the *fsync* system call is invoked. Specifically, noseFS maintains a list of the updated *extents* of a file in DRAM, and synchronizes the updates to NVMs in the *fsync* operation.

In the *fsync* system call, noseFS first allocates and initializes the new *extents* in NVMs. Then it records the updates to the log. After making the new *extents* and log entries durable, noseFS atomically updates the log tail. Finally, it updates the *extents* list and the bitmap in NVMs, and atomically updates the log tail to clean the log.

Memory-mapped I/O noseFS proposes the lightweight *atomic-mmap* to provide consistency for the memory-mapped files and reduce the redundant copy overhead. When the *mmap* operation is invoked, it allocates new data blocks from NVMs and copies the file data to the new space. Then it maps the new blocks into the user address space. When the *msync* operation is invoked, it uses NCOW and the lightweight journaling to atomically copy the data from the memory-mapped blocks back to the file, as the *write* and *fsync* operations do.

To improve performance, noseFS tracks the process of the page fault interrupt when applications access memory-mapped files. It records the modified blocks and sets the dirty flags. When the *msync* operation is invoked, noseFS synchronizes only the dirty blocks instead of the whole memory-mapped blocks to NVMs, and cleans the dirty flags of the blocks. If there are no updates to these blocks before the next *msync*, they will not be synchronized again.

E. Crash Consistency

A file system operation is atomic if and only if updates made by the operation are committed in all or none. noseFS supports the atomicity for the common file system operations. Specifically, it adopts the 8-byte atomic updates technique as previous work [6], [7], [12] to guarantee the atomicity for the updates of no more than 8 bytes, and utilizes its lightweight journaling technique to atomically log the updates of more than 8 bytes. Then the file system can survive and recover to maintain consistency upon crashes or power failures.

8-byte atomic updates Modern processors support 8-byte atomic writes for DRAM, and the systems [6], [7] assume that 8-byte writes to NVMs are atomic as well. noseFS uses the 8-byte atomic in-place write to update the *pinode*'s access time for *read* and the 8-byte log tail for journaling.

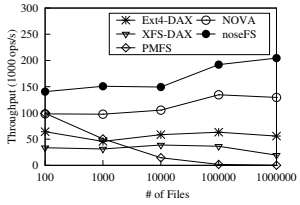
Lightweight journaling noseFS adopts redo journaling to support the recovery of file system operations, and it consists of the following four steps. First, it allocates new data blocks, *pinodes* and *extents* with invalid bits in NVMs to write the new data, and logs the updates before updating the log tail. Second, it atomically updates the log tail with the amount of committed log entries to log the updates before committing updates to the file system. Third, it updates the associated data structures (i.e., the global hash table, the *extents* list of a file, and the bitmap), and sets the valid bits of *pinodes* and *extents* in NVMs before recycling the old data. Finally, it cleans the log by atomically updating the log tail with 0 (i.e., there are no log entries) to reduce the log size and simplify the recovery, and then recycles the old data. In addition, the write ordering of the above steps is guaranteed by orderly flushing the CPU cacheline with *CLFLUSH* and *SFENCE* instructions.

Recovery As described above, the associated data structures (i.e., the global hash table, the *extents* list of a file and the bitmap) and the valid bits of *pinodes* and *extents* in NVMs are updated to the file system only after atomically committing the entire log entries successfully. If a crash happens before finishing the logging of a file system operation, the file system in NVMs keeps consistent without being affected. If a crash happens after the atomic update of a log tail, noseFS reads the log entries in the directory journal and file journal, respectively, to redo the updates. If a crash happens after cleaning the log, all updates have been persisted.

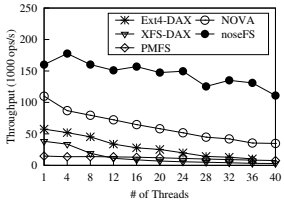
For metadata recovery, noseFS first reads the log entries recorded in the directory journal. Then it updates the hash table in NVMs, and cleans the directory journal. Finally, it rebuilds the mirror hash table in DRAM according to the hash table in NVMs. For file data recovery, noseFS first reads the log entries recorded in the file journal. Then it updates the bitmap in NVMs and rebuilds the *extent* list in NVMs, and then cleans the file journal. Finally, it rebuilds the file index tree in DRAM according to the *extent* list in NVMs.

IV. Experimental Study

In this section, we evaluate the performance and scalability of noseFS using a set of micro and macro benchmarks aiming at answering the following questions. (1) Does noseFS perform

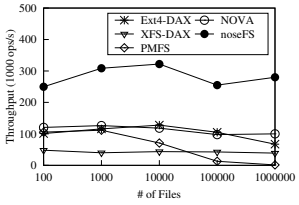


(a) single thread

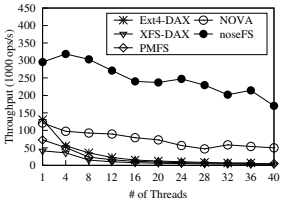


(b) multiple threads

Fig. 4. Average throughput of each thread performing *create* operations.

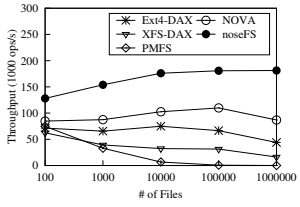


(a) single thread

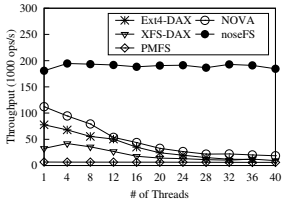


(b) multiple threads

Fig. 5. Average throughput of each thread performing *unlink* operations.

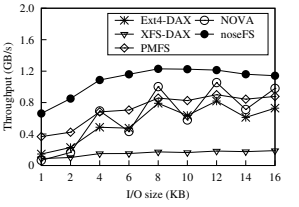


(a) single thread

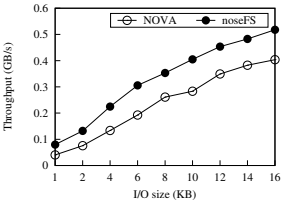


(b) multiple threads

Fig. 6. Average throughput of each thread performing *rename* operations.



(a) append



(b) random write

Fig. 7. Throughput of *write* operations (single thread).

better than the state-of-the-art file systems designed for disks and NVMs? (2) What are the performance benefits of noseFS? (3) How is the scalability of noseFS compared with the existing file systems on many-core platforms?

A. Experimental Setup

We evaluate noseFS on Linux kernel 4.6.5 against four file systems: NOVA [6], PMFS [7], Ext4-DAX [13] and XFS-DAX [14], and the experiments of the four file systems are on the original Linux kernel 4.6.5 without noseFS checking. NOVA and PMFS are the open-source file systems that are specifically designed for NVMs. Ext4-DAX(*ordered*) and XFS-DAX are the traditional file systems designed for disk. NOVA is the only file system in the group providing the same strong consistency guarantees as noseFS for both metadata and data. The others only guarantee metadata consistency and perform in-place updates for file data. We mainly compare the performance and scalability with NOVA.

The used Intel Xeon E7 server is configured with 4 processors with 2.2GHz, and each processor has 10 cores. The emulation of NVMs is based on a 150G DRAM that is treated by OS as a persistent memory region.

B. Experimental Results with Micro-benchmarks

We use three micro-benchmarks: *dirOps*, *fileOps* and *fio* [15] to evaluate the performance and scalability of noseFS. The *dirOps* and *fileOps*, the benchmarks that we created, create multiple threads (from 1 to 40), respectively, and perform the corresponding operations in parallel.

(1) *micro-benchmark dirOps*

dirOps is to evaluate the effect of the lightweight naming and the lightweight journaling. It contains three directory operations: *create*, *rename* and *unlink*. For single thread benchmark, it creates a fixed number of files in one directory, and then renames the files in the same directory, and finally deletes all of them. We vary the number of files from 100 to 1000000 to evaluate the sensitivity of different directory structures. For multiple threads benchmark, each thread handles 10000 files.

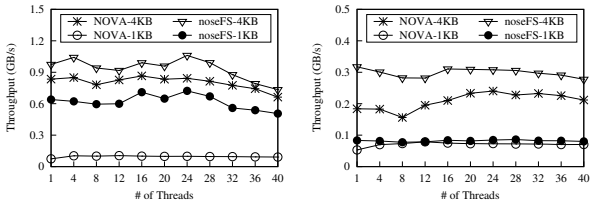
***create* operations** Figure 4(a) shows that noseFS outperforms NOVA by 0.42X to 0.58X in single thread, and Figure 4(b) shows that both noseFS and NOVA support high concurrent *create* operations. The improvement is mainly due to the allocation and initialization of single *pinode*, and its insertion of the efficient global hash table. While for other file systems, they allocate and initialize both the *inode* and the *dentry*, and insert them into *dcache* and *icache* respectively.

***unlink* operations** Figure 5(a) shows that noseFS outperforms NOVA by 1.07X to 1.79X in single thread, and Figure 5(b) shows that both noseFS and NOVA support high concurrent *unlink* operations. The improvement is mainly because of the removal and destruction of single *pinode* instead of both *inode* and *dentry*. In addition, NOVA commits the updates to both directory log and file log, while noseFS only logs the update to directory journal.

***rename* operations** Figure 6(a) shows that noseFS outperforms NOVA by 0.51X to 1.09X in a single thread, and it is mainly because of the removal and insertion of both *inodes* and *dentries*. Figure 6(b) shows that only noseFS supports high concurrent *rename* operations because of the fine-grained *seqlocks* (local lock), and all the other file systems encounter the scalability bottleneck because of the global *rename_lock*.

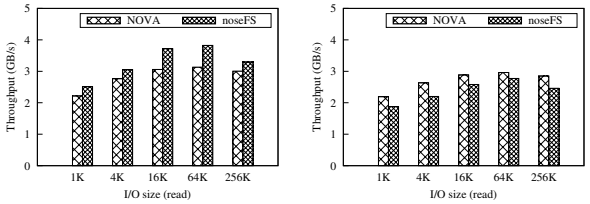
(2) *micro-benchmark fileOps*

fileOps is to evaluate the effect of NCOW and the lightweight journaling. It contains three file operations: *append*, *random write* and *read*. For single thread benchmark, it appends to a file (the total write size is 16GB), and then randomly overwrites the file (the random offsets of the 16GB file are in byte granularity). We alter the I/O size (i.e., the amount of data written per system call) from 1KB to 16KB to compare the overhead of copy and *extent* fragments. Note that we perform the *fsync* operation for noseFS after performing each *write* operation, to synchronize the updates and guarantee the same data consistency as NOVA. For multiple threads benchmark, each thread writes to a 512MB file with I/O sizes 1KB and 4KB, respectively. In order to evaluate how *extent*



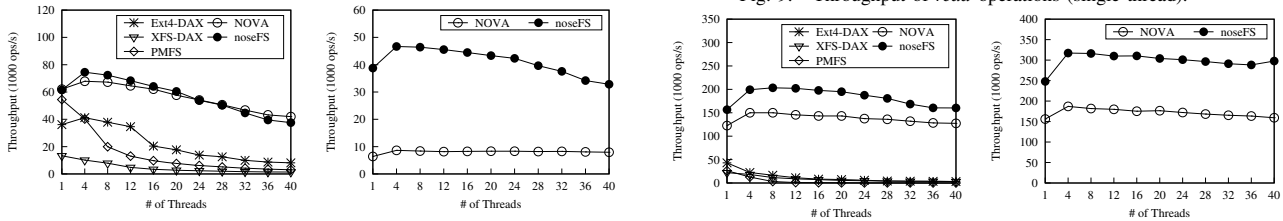
(a) append (b) random write

Fig. 8. Throughput of *write* operations (multiple threads).



(a) read after append (b) read after random write

Fig. 9. Throughput of *read* operations (single thread).



(a) Fileserver (b) Fileserver-1K (c) Varmail (d) Varmail-1K

Fig. 10. Average throughput of each thread performing Filebench.

fragments affect *read* operation, the single thread benchmark also reads the 16GB file after appending and overwriting it with 4KB I/O size. We alter the read size from 1KB to 256KB.

***write* operations in single thread** Figures 7 shows that noseFS provides the highest throughput. It outperforms NOVA by 0.15X (with 12KB I/O size) to 9.14X (with 1KB I/O size) for *append*, and 0.26X (with 14KB I/O size) to 0.97X (with 1KB I/O size) for *random write*. This is due to *NCOW* that eliminates the copy overhead, and the lightweight journaling that reduces the redundant metadata write. Specifically, as can be seen in Figure 7(a), the 4KB-block-unaligned *write* operations (with 1KB, 2KB, 6KB, 10KB and 14KB I/O sizes) bring the obviously higher performance improvement than the other 4KB-block-aligned *write* operations.

TABLE I
THROUGHPUT OF MEMORY-MAPPED I/O (IN MB/S).

	1M	4M	8M	12M
NOVA	7.351	2.041	1.031	0.694
noseFS	204.8	250.98	303.907	271.087

In noseFS, the write offset and I/O size affect the scale of the file index tree. *Random write* operations with small I/O size bring a large number of *extent* fragments, which results in the growth of the tree size and further leads to slow search, insertion and removal. However, according to Figure 7(b) (with 1KB I/O size), the redundant copy overhead remains larger than the overhead introduced by the heavy file index tree.

***write* operations in multiple threads** Figures 8 shows that both noseFS and NOVA provide good scalability for the *append* and *random write* operations.

***read* operations in single thread** Figure 9 shows that the read performance of noseFS is approximately the same as NOVA, and it is slightly lower than NOVA when there are plenty of *extent* fragments as shown in Figure 9(b).

(3) *micro-benchmark fio*

fio is to evaluate the effort of the lightweight *atomic-mmap*. We alter the file size from 1MB to 12MB. For each file, we use *fio* to perform writes to the memory-mapped region with a 4KB I/O size, and then use *msync* to synchronize the whole file after performing each *write* operation.

Table I shows the total throughput of the *write* and *msync* operations (memory-mapped I/O). noseFS provides 26.9X to 389.6X throughput improvements compared with NOVA, which provides the same *mmap* data consistency guarantees as noseFS. The improvement is because of the lightweight *atomic-mmap* that noseFS writes only the dirty pages to file data when the system call *msync* is invoked.

C. Experimental Results with Macro-benchmarks

Filebench [16] is a file system benchmark that can simulate a variety of complex workloads by specifying different models. We use Varmail and Fileserver, which are write-intensive workloads, to evaluate the performance of noseFS for real world applications. Table II summarizes the characteristics of the workloads. We run these benchmarks in each thread and there are 1 to 40 threads performing concurrently.

TABLE II
FILEBENCH WORKLOAD CHARACTERISTICS.

Workload	Average file size	# of files	I/O size	R/W ratio
Fileserver	128KB	5K	1MB	1:2
Fileserver-1K	128KB	5K	1KB	1:2
Varmail	16KB	50K	16KB	1:1
Varmail-1K	16KB	50K	1KB	1:1

(1) *macro-benchmark Fileserver*

The average file size in Fileserver is 128KB, in which the data write overhead dominates. As shown in Figure 10, noseFS provides the highest throughput and achieves high scalability. For the results shown in Figure 10(a), the throughput of noseFS is approximately same as NOVA when performing the 4KB-block-aligned *writes*. For the results shown in Figure 10(b), noseFS outperforms NOVA by 3.1X to 5.1X with 1KB write size, which is mainly contributed by *NCOW* and the lightweight journaling techniques.

(2) *macro-benchmark Varmail*

The Varmail acts as a mail server, and both directory and file operations contribute to the performance improvements.

For the workload performing 4KB-block-aligned *writes* shown in Figure 10(c), noseFS outperforms NOVA by 0.25x to 0.36x because of the faster *create* and *unlink* operations. For the workload performing 4KB-block-unaligned *writes* shown in Figure 10(d), noseFS outperforms NOVA by 0.59x to 0.87x. As the file size is only 16KB in Varmail, the benefit of the *write* operation is not as obvious as Fileserver.

V. Related Work

NVM-based file systems. Existing NVMFSs are highly optimized around the advanced properties of NVMs such as byte-addressability. BPFS [5] proposes the short-circuit shadowing paging to reduce the copy-on-write overhead for maintaining consistency. SCMFS [17] utilizes the memory management module and maps files to contiguous virtual address space, making its implementation simplified and efficient. PMFS [7] avoids the block layer and adopts fine-grained logging combined with atomic updates for consistency. Aerie [9] offers the file system interface in user space to provide higher performance than a kernel implementation. NOVA [6] adopts log-structuring to efficiently provide strong consistency for conventional file operations and mmap-based access. NOVA-Fortis [18] adds fault-tolerance to NOVA by taking consistent snapshots. HiNFS [19] buffers the lazy-persistent writes in DRAM temporarily to hide the long write latency of N-VMs. SoupFS [8] revisits soft update and proposes pointer-based dual views to guarantee the correctness and consistency without synchronous cache flushes and page cache. However, different from noseFS, they do not fully exploit the NVMs characteristics, and still need the conventional VFS to provide in-memory cache for file system objects.

Directory organization. Most existing NVMFSs pay close attention to directory operations due to their significant impacts on application performance [20]–[22]. NOVA [6] keeps a radix tree in DRAM for each directory *inode* to speed up the *dentry* lookups. SoupFS [8] uses hash tables to re-organize directories, simplifying the dependencies caused by block-oriented directory organization. However, they still improve the performance within the confinement of the traditional I/O stack architecture, and use conventional disk-oriented VFS to provide hierarchical naming and access protection. In contrast, noseFS integrates conventional VFS and the namespace of underlying file systems into a single lightweight naming within the I/O stack for NVMs.

NVM logging. As emerging NVMs are expected to offer DRAM-like performance and data persistence, many recent studies propose to deploy NVMs to perform database logging [23]–[25]. Wang et al. [23] propose a distributed logging to provide fast, scalable logging performance on many-core platforms. The proposed distributed logging uses logical clocks to track and resolve transaction dependencies among multiple logs and update records. Our work also adopts the distributed journaling mechanism to avoid global contention and improve concurrency. However, in contrast to tracking transaction dependencies, our work adopts scalable, persistent data structures to eliminate dependencies between updates.

VI. Conclusions

In this paper, we present noseFS, a highly scalable and efficient file system for NVMs. noseFS redesigns the lightweight naming by organizing metadata structures with the global hash table. It achieves the Non-Copy-On-Write by using the fine-grained byte-unit file index tree, and the lightweight journaling logs the address of the metadata only, reducing redundant write overhead introduced by the data consistency guarantee. It also provides the lightweight *atomic-mmap* to provide the consistency guarantee for memory-mapped files with low overhead. Evaluation results show that noseFS outperforms existing NVMFSs with equally strong consistency guarantees, and achieves near-linear scalability on a 40-core machine.

References

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro 30 (Jan. 2010)*, pp. 131–141, 2010.

[2] M. E. Ali Ahari and, F. Oboril, and M. Tahoori, "Improving reliability, performance, and energy efficiency of stt-mram with dynamic write latency," in *ICCD*, 2015.

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *ASPLOS*, 2011.

[4] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *MSST*, 2015.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.

[6] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST*, 2016.

[7] D. S. Rao, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys*, 2014.

[8] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *ATC*, 2017.

[9] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: flexible file-system interfaces to storage-class memory," in *EuroSys*, 2014.

[10] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "Multilanes: A providing virtualized storage for os-level virtualization on many cores," in *FAST*, 2014.

[11] "Path lookup in linux kernel," https://lwn.net/Articles/419826/.

[12] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST*, 2015.

[13] "Support ext4 on nv-dimms," https://lwn.net/Articles/609652/.

[14] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, , and G. Peck, "Scalability in the xfs file system," in *ATC*, 1996.

[15] "Flexible io (fio) tester," http://freecode.com/projects/fio.

[16] "File system benchmark," https://github.com/filebench/filebench/wiki.

[17] X. Wu and A. Reddy, "Scmfs : A file system for storage class memory," in *SC*, 2011.

[18] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudof, "NOVA-Fortis: A fault-tolerant non-volatile main memory file system," in *SOSP*, 2017.

[19] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *EuroSys*, 2016.

[20] P. H. Lensing, T. Cortes, and A. Brinkmann, "Direct lookup and hash-based metadata placement for local file systems," in *SYSTOR*, 2013.

[21] Y. Lu, J. Shu, and W. Wang, "Reconfs: A reconstructable file system on flash storage," in *SYSTOR*, 2013.

[22] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *SOSP*, 2015.

[23] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *PVLDB*, vol. 7, no. 10, pp. 865–876, 2014.

[24] J. Arulraj, A. Pavlo, and S. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *SIGMOD*, 2015.

[25] W. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: exploiting NVRAM in write-ahead logging," in *ASPLOS*, 2016.