SpanFS: A Scalable File System on Fast Storage Devices

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai, *Beihang University*

https://www.usenix.org/conference/atc15/technical-session/presentation/kang

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

SpanFS: A Scalable File System on Fast Storage Devices

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma and Jinpeng Huai

SKLSDE Lab, Beihang University, China

{*kangjb, woty, yuwr, dulian, mashuai*}*@act.buaa.edu.cn*, *zblgeqian@gmail.com*, *huaijp@buaa.edu.cn*

Abstract

Most recent storage devices, such as NAND flash-based solid state drives (SSDs), provide low access latency and high degree of parallelism. However, conventional file systems, which are designed for slow hard disk drives, often encounter severe scalability bottlenecks in exploiting the advances of these fast storage devices on many-core architectures. To scale file systems to many cores, we propose SpanFS, a novel file system which consists of a collection of micro file system services called domains. SpanFS distributes files and directories among the domains, provides a global file system view on top of the domains and maintains consistency in case of system crashes.

SpanFS is implemented based on the Ext4 file system. Experimental results evaluating SpanFS against Ext4 on a modern PCI-E SSD show that SpanFS scales much better than Ext4 on a 32-core machine. In micro-benchmarks SpanFS outperforms Ext4 by up to 1226%. In application-level benchmarks SpanFS improves the performance by up to 73% relative to Ext4.

1 Introduction

Compared to hard disk drives (HDDs), SSDs provide the opportunities to enable high parallelism on many-core processors [9, 15, 29]. However, the advances achieved in hardware performance have posed challenges to traditional software [9, 27]. Especially, the poor scalability of file systems on many-core often underutilizes the high performance of SSDs [27].

Almost all existing journaling file systems maintain consistency through a centralized journaling design. In this paper we focus on the scalability issues introduced by such design: (1) The use of the centralized journaling could cause severe contention on *in-memory* shared data structures. (2) The transaction model of the centralized journaling serializes its internal I/O actions *on devices* to ensure correctness, such as committing and checkpointing. These issues will sacrifice the high parallelism provided by SSDs. An exhaustive analysis of the scalability bottlenecks of existing file systems is presented in Section 2 as the motivation of our work.

Parallelizing the file system service is one solution to file system scalability. In this paper, we propose SpanFS, a novel journaling file system that replaces the centralized file system service with a collection of independent micro file system services, called *domains*, to achieve scalability on many-core. Each domain performs its file system service such as data allocation and journaling independently. Concurrent access to different domains will not contend for shared data structures. As a result, SpanFS allows multiple I/O tasks to work in parallel without performance interference between each other.

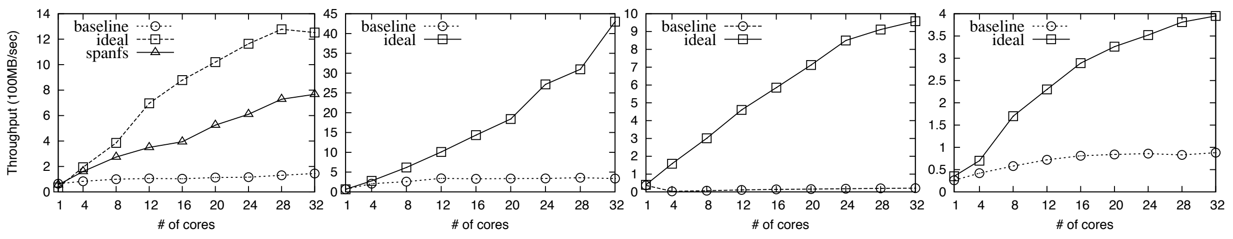
Apart from performance, consistency is another key aspect of modern file systems. Since each domain is capable of ensuring the consistency of the on-disk structures that belong to it, the key challenge to SpanFS is to maintain crash consistency on top of multiple domains. SpanFS proposes a set of techniques to distribute files and directories among the domains, to provide a global file system view on top of the domains and to maintain consistency in case of system crashes.

We have implemented SpanFS based on the Ext4 file system in Linux kernel 3.18.0 and would demonstrate that SpanFS scales much better than Ext4 on 32 cores, thus bringing significant performance improvements. In micro-benchmarks, SpanFS outperforms Ext4 by up to 1226%. In application-level benchmarks SpanFS improves the performance by up to 73% relative to Ext4.

The rest of the paper is organized as follows. Section 2 analyzes the scalability issues of existing file systems. Section 3 presents the design and implementation of SpanFS. Section 4 shows the performance result of SpanFS. We relate SpanFS to previous work in Section 5 and present the conclusion and future work in Section 6.

2 Background and Motivation

Most modern file systems scale poorly on many-core processors mainly due to the *contention* on shard data structures in memory and *serialization* of I/O actions on device. Our previous work [27] has identified some lock bottlenecks in modern file systems. We now provide an in-depth analysis of the root causes of the poor scalability. We first introduce the file system journaling mechanism to facilitate the scalability analysis. Then, through a set of experiments we will 1) show the scalability issues in existing modern file systems, 2) identify the scalability bottlenecks and 3) analyze which bottlenecks can be eliminated and which are inherent in the centralized file system design.



(a) Ext4 (b) XFS (c) Btrfs (d) ZFS

Figure 1: **Scalability Evaluation.** *We carry out this experiment in Linux 3.18.0 kernel on a RAM disk. We preallocate all the pages of the RAM disk to avoid contention within the RAM disk for the baselines and SpanFS. The total journaling sizes of SpanFS, Ext4, XFS and the ideal file systems based on Ext4 and XFS are all set to 1024 MB, respectively. For ZFS, we compile the recently released version (0.6.3) on Linux. This test is performed on a 32-core machine. For some lines the better than linear speedup is probably due to the Intel EIST technology.*

Ext4				**XFS**			
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)	Percent	Lock Name	Bounces	Total Wait Time (Avg. Wait Time)	Percent
journal->j_wait_done_commit	11845 k	1293 s (103.15 µs)	27%	cil->xc_push_lock	8019 k	329 s (37.26 µs)	13.8%
journal->j_list_lock	12713 k	154 s (11.34 µs)	3.2%	iclog->ic_force_wait	2188 k	87.4 s (39.94 µs)	3.7%
journal->j_state_lock-R	1223 k	7.1 s (5.19 µs)	0.1%	cil->xc_ctx_lock-R	1136 k	80.1 s (70.02 µs)	3.4%
journal->j_state_lock-W	956 k	4.3 s (4.29 µs)	0.09%	pool->lock	3673 k	34.1 s (9.28 µs)	1.4%
zone->wait_table	925 k	3.1 s (3.36 µs)	0.06%	log->l_icloglock	1555 k	25.8 s (16.18 µs)	1%

Table 1: **The top 5 hottest locks.** *We show the top 5 hottest locks in the I/O stack when running 32 Sysbench instances. These numbers are collected in a separated kernel with lock stat compiled. As* lock stat *introduces some overhead, the numbers does not accurately represent the lock contention overhead in Figure 1. "Bounces" represents the number of lock bounces among CPU cores. We calculate the percent of the lock wait time in the total execution time by dividing the lock wait time divided by the number of instances (32) by the total execution time.*

2.1 File System Journaling

Our discussion is based on the Ext3/4 journaling mechanism [39], which adopts the group commit mechanism [23] for performance improvements. Specifically, there is only one running transaction that absorbs all updates and at most one committing transaction at any time [39]. As a result, one block that is to be modified in the OS buffer does not need to be copied out to the journaling layer unless that block has already resided within the committing transaction, which largely reduces the journaling overhead caused by dependency tracking [39].

Ext4 [13] adopts JBD2 for journaling. For each update operation Ext4 starts a JBD2 handle to the current running transaction to achieve atomicity. Specifically, Ext4 passes the blocks (refer to metadata blocks in ordered journaling mode) to be modified associated with the handle to the JBD2 journaling layer. After modifying these blocks, Ext4 stops the handle and then the running transaction is free to be committed by the JBD2 journaling thread. These modified block buffers will not be written back to the file system by the OS until the running transaction has been committed to the log [39]. For simplicity, we refer to the above process as wrapping the blocks to be modified in a JBD2 handle in the rest of the paper.

2.2 Scalability Issue Analysis

We use Sysbench [1] to generate update-intensive workloads to illustrate the scalability bottlenecks. Multiple single-threaded benchmark instances run in parallel, each of which issues 4KB sequential writes and invokes *fsync()* after each write. Each instance operates over 128 files with a total write traffic of 512MB. We vary the number of running instances from 1 to 32 and the number of used cores is euqal to the number of instances. We measure the total throughput.

Four file systems are chosen as baseline for analysis: Ext4, XFS [38], Btrfs [35] and OpenZFS [2], of which Ext4 and XFS are journaling file systems while Btrfs and ZFS are copy-on-write file systems. An ideal file system is set up by running each benchmark instance in a separated partition (a separated RAM disk in this test) managed by the baseline file system, which is similar to the disk partition scenario in [31]. It is expected to achieve linear scalability since each partition can perform its file system service independently.

Figure 1 shows that all the four baseline file systems scale very poorly on many-core, resulting in nearly horizontal lines. The "ideal" file systems exhibit near-linear scalability. We add the result of SpanFS with 16 domains in Figure 1(a), which brings a performance improvement of 4.29X in comparison with the stock Ext4 at 32 cores.

2.3 Scalability Bottleneck Analysis

To understand the sources of the scalability bottlenecks, we collect the lock contention statistics using *lock stat* [7]. Due to space limitation, we show the statistics on top

SpanFS-16		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
journal->j_wait_done_commit	3333 k	38.5 s (11.13 μs)
journal->j_state_lock-R	4259 k	16.6 s (3.70 μs)
journal->j_state_lock-W	2637 k	10.5 s (3.84 μs)
journal->j_list_lock	5042 k	10.2 s (2.00 μs)
zone->wait_table	226 k	0.5 s (2.06 μs)

Table 2: **The top 5 hottest locks.** *The top 5 hottest locks in the I/O stack when running the sysbench benchmark on SpanFS.*

hottest locks of Ext4 and XFS in Table 1. Table 2 shows the top 5 hottest locks when running the same benchmark on SpanFS at 32 cores. Ext4 and XFS spend substantial wait time acquiring hot locks and the average wait time for these hot locks is high. In contrast, SpanFS reduces the total wait time of the hot locks by around 18X (76 s vs 1461 s).

Btrfs also has a few severely contended locks, namely eb->write_lock_wq, btrfs-log-02 and eb->read_lock_wq. The total wait time of these hot locks can reach as much as 14476 s, 5098 s and 2661 s respectively. We cannot collect the lock statistics for ZFS using *lock stat* due to the license compatible issue.

2.3.1 Contention on shared data structures

Now we look into Ext4 and discuss the causes of scalability bottlenecks in depth, some of which are also general to other file systems. As is well known, shared data structures can limit the scalability on many-core [10, 11]. JBD2 contains many shared data structures, such as the journaling states, shared counters, shared on-disk structures, journaling lists, and wait queues, which can lead to severe scalability issues.

(a) Contention on the journaling states. The journaling states are frequently accessed and updated, and protected by read-write lock (i.e., *j_state_lock*). The states may include the log tail and head, the sequence numbers of the next transaction and the most recently committed transaction, and the current running transaction's state. The lock can introduce severe contention. The RCU lock [33] and Prwlock [30] are scalable for read-mostly workloads while JBD2 have many writes to these shared data structures in general as shown in Table 1. Hence, they are not effective to JBD2.

(b) Contention on the shared counters. The running transaction in JBD2 employs atomic operations to serialize concurrent access to shared counters, such as the number of current updates and the number of buffers on this transaction, which can limit the scalability. Sometimes, JBD2 needs to access the journaling states and these shared counters simultaneously, which can cause even more severe contention. For instance, to add updates to the running transaction, JBD2 needs to check whether there is enough log free space to hold the running transaction by reading the number of the buffers on the running transaction and on the committing transaction and reading the log free space. We have confirmed the contention on shared counters using *perf*, which will partly cause the JBD2 function *start_this_handle* to account for 17% of the total execution time when running 32 Filebench Fileserver instances. Adopting per-core counters such as sloppy counter [11] and Refcache [18] will introduce expensive overhead when reading the true values of these counters [18].

(c) Contention on the shared on-disk structures. Although the on-disk structures of Ext4 are organized in the form of block groups, there is also contention on shared on-disk structures such as block bitmap, inode bitmap and other metadata blocks during logging these blocks. These were not manifested in Table 1 since *lock stat* does not track the bit-based spin locks JBD2 uses.

(d) Contention on the journaling lists. JBD2 uses a spin lock (i.e., *j_list_lock*) to protect the transaction buffer lists and the checkpoint transaction list that links the committed transactions for checkpointing, which can sabotage scalability. Replacing each transaction buffer list with per-core lists may be useful to relieve the contention. However, using per-core lists is not suitable for the checkpoint transaction list as JBD2 needs to checkpoint the transactions on the list in the order that the transactions are committed to the log.

(e) Contention on the wait queues. JBD2 uses wait queues for multi-thread cooperation among client threads and the journaling thread, which will cause severe contention when a wait queue is accessed simultaneously. The wait queue is the most contended point in Ext4 during our benchmarking. Simply removing this bottleneck, i.e. per-core wait queue [30], cannot totally scale Ext4 as other contention points will rise to become the main bottlenecks, such as *j_state_lock* and shared counters. The most contended point in XFS is not the wait queue as shown in Table 1. Hence, we need a more thorough solution to address the scalability issues.

2.3.2 Serialization of internal actions

The centralized journaling service usually needs to serialize its internal actions in right order, which also limits the scalability. Here we give two examples.

Parallel commit requests are processed sequentially in transaction order by the journaling thread [39], which largely sacrifices the high parallelism provided by SSDs. Enforcing this order is necessary for correctness when recovering the file system with the log after a crash due to the dependencies between transactions [16].

Another example is when the free space in the log is not enough to hold incoming update blocks. JBD2 performs a checkpoint of the first committed transaction on the checkpoint transaction list to make free space. Parallel checkpoints also have to be serialized in the order that

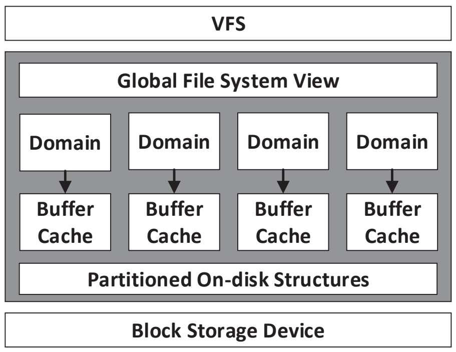


Figure 2: **SpanFS Architecture.**

transactions are committed to the log.

2.3.3 Summary

We analyzed the scalability bottlenecks of Ext4, which are mainly caused by the contention on the shared data structures in memory and the serialization of internal actions on devices. The use of shared data structures are inherent in the centralized journaling design. The serialization of journaling would also need to access and update the shared data structures, such as log head and tail.

To address the scalability bottlenecks, file systems should be restructured to reduce the contention on shared data structures and to parallelize the file system service.

3 Design and Implementation

We present the design and implementation of SpanFS and introduce the key techniques to provide a global file system view and crash consistency.

3.1 SpanFS Architecture

Figure 2 shows the architecture of SpanFS, which consists of multiple micro file system services called domains. SpanFS distributes files and directories among multiple domains to reduce contention and increase parallelism within the file system. Each domain has its independent on-disk structures, in-memory data structures and kernel services (e.g., the journaling instance JBD2) at runtime. As there is no overlap among the domains' on-disk blocks, we allocate a dedicated buffer cache address space for each domain to avoid the contention on the single device buffer cache. As a result, concurrent access to different domains will not cause contention on shared data structures. Each domain can do its journaling without the need of dependency tracking between transactions and journaled buffers that belong to different domains, enabling high parallelism for logging, committing and checkpointing.

SpanFS provides a global file system view on top of the domains by building global hierarchical file system namespace and also maintains global consistency in case of system crashes.

3.2 Domain

The domain is the basic independent function unit in SpanFS to perform the file system service such as data allocation and journaling. During mounting, each domain will build its own in-memory data structures from its on-disk structures and start its kernel services such as the JBD2 journaling thread. In the current prototype SpanFS builds the domains in sequence. However, the domains can be built in parallel by using multiple threads.

3.2.1 SpanFS on-disk layout

In order to enable parallel journaling without the need of dependency tracking, we partition the device blocks among the domains. SpanFS creates the on-disk structures of each domain on the device blocks that are allocated to the domain. The on-disk layout design of each domain is based on the Ext4 disk layout [5]: each domain mainly has a super block, a set of block groups, a root inode and a JBD2 journaling inode (i.e., log file).

Initially, the device blocks are evenly allocated to the domains. Our architecture allows to adjust the size of each domain online on demand in the unit of block groups. Specifically, the block groups in one domain can be reallocated to other domains on demand. To this end, we should store a block group allocation table (BAT) and a block group bitmap (BGB) on disk for each domain. The BGB is maintained by its domain and is used to track which block group is free by the file system. When the free storage space in one domain drops to a predefined threshold, the file system should reallocate the free block groups in other domains to this domain. To avoid block group low utilization each domain should allocate inodes and blocks from the block groups that have been used as far as possible. As the global block group reallocation can cause inconsistent states in case of crashes, we should create a dedicated journaling instance to maintain the consistency of the BATs. Each domain should first force the dedicated journaling to commit the running transaction before using the newly allocated block groups. This ensures the reallocation of block groups to be persisted on disk and enables the block groups to be correctly allocated to domains after recovery in case of crashes. We leave the implementation of online adjusting of each domain's size on demand as our future work.

In our current implementation SpanFS only supports static allocation of block groups. Specifically, we statically allocate a set of contiguous blocks to each domain when creating the on-disk structures by simply storing the first data block address and the last data block address in each domain's super block. Each domain's super block is stored in its first block group. In order to load all the domains' super blocks SpanFS stores the next super block address in the previous super block. Each domain adopts the same policy as Ext4 for inode and block allocation among its block groups.

3.2.2 Dedicated buffer cache address space

The Linux operating system (OS) adopts a buffer cache organized as an address space radix tree for each block device to cache recently accessed blocks and use a spin lock to protect the radix tree from concurrent inserts. Meanwhile, the OS uses another spin lock to serialize concurrent access to each cache page's buffer list. As a result, when multiple domains access the single underlying device simultaneously, the above two locks will be contended within the buffer cache layer .

The device block range of each domain does not overlap with those of other domains and the block size is the same with the page size in our prototype. Concurrent accesses to different domains should not be serialized in the buffer cache layer as they can commute [19, 18].

We leverage the Linux OS block device architecture to provide a dedicated buffer cache address space for each domain to avoid lock contention. The OS block layer manages I/O access to the underlying device through the block device structure, which can store the inode that points to its address space radix tree and pointers to the underlying device structures such as the device I/O request queue and partition information if it is a partition. SpanFS clones multiple block device structures from the original OS block device structure and maps them to the same underlying block device. SpanFS assigns a dedicated block device structure to each domain during mounting so that each domain can have its own buffer cache address space.

Under the block group reallocation strategy, the device block range of each domain may be changed over time. We should remove the pages in the domain's buffer cache address space corresponding to the block groups that are to be reallocated to other domains. This process can be implemented with the help of the OS interface (*invalidate_mapping_pages()*). We leave the implementation of buffer cache address space adjusting as our future work.

3.3 Global Hierarchical Namespace

In order to distribute the global namespace, SpanFS chooses a domain as the root domain to place the global root directory and then scatters the objects under each directory among the domains. Specifically, SpanFS distributes the objects (files and directories) under each directory using a round-robin counter for the directory. The use of per directory counter can avoid global contention.

To support this distribution, SpanFS introduces three types of directory entries (dentries): normal dentry, shadow dentry and remote dentry, as illustrated in Figure 3. We call an object placed in the same domain with its parent directory a normal object. The domain where the parent directory lies is referred to as the local domain. SpanFS creates an inode and a *normal dentry* pointing

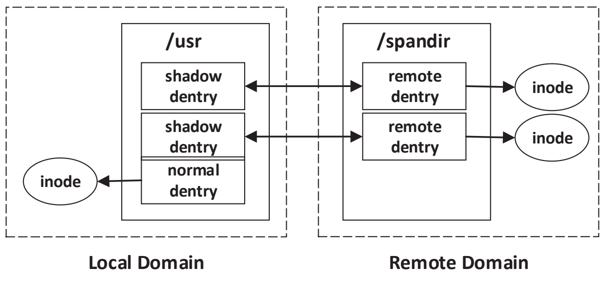


Figure 3: **The connection between domains.** *This figure illustrates how SpanFS distributes a object to a remote domain.*

to this inode under the parent directory in the local domain. We call an object placed in the different domain from its local domain a distributed object, the different domain is referred to as remote domain. SpanFS creates an inode and a *remote dentry* pointing this inode in the remote domain and then creates a *shadow dentry* under its parent directory in the local domain for look up. The reason why SpanFS creates a remote dentry and an inode rather than only an inode is due to consideration of maintaining global consistency, which will be described in Section 3.4. In this paper, the dentry and inode refer to the on-disk structures rather than the VFS structures.

To store remote dentries from other domains, each domain creates a set of special directories called *span directories* which are invisible to users. The number of span directories in each domain is set to 128 by default and the span directories are evenly allocated to each CPU core. When a thread is to create a remote dentry, SpanFS selects a span directory randomly from the ones allocated to the CPU core.

SpanFS constitutes the *bidirectional index* by embedding enough information in both the shadow dentry and remote dentry to make them point to each other. For each shadow dentry, SpanFS stores in the shadow dentry fields the remote dentry's domain ID, the inode number of the span directory where the remote dentry stays, the remote inode number and the name. For each remote dentry, SpanFS stores the shadow dentry's domain ID, the inode number of the parent directory where the shadow dentry lies, the remote inode number and the name.

3.4 Crash Consistency Model

As each domain is capable of ensuring the consistency of its on-disk structures in case of system crashes through journaling, the most critical challenge to SpanFS is how to maintain global consistency across multiple domains.

The following example will demonstrate the consistency challenges. To create a distributed object, one approach is to create an inode in the remote domain, and to add a dentry in the local domain which points to the remote inode, which can result in two possible inconsistent states in case of crashes. The first case is that the inode reaches the device but the dentry is lost. Then the newly created inode becomes storage garbage silently, which cannot be reclaimed easily. Another case is that the dentry reaches the device but the inode fails. As a result the dentry points to a non-existent inode.

Another approach to address the above problem is adding dependency information to the running transactions of the two domains. However, this approach will end up with many dependencies between multiple domains, with the cost of dependency tracking and serializing the journaling actions among the domains.

Based on *bidirectional index*, SpanFS adopts two mechanisms *stale object deletion* and *garbage collection* to address the problem. As discussed in Section 3.3, SpanFS creates a remote dentry and inode in the remote domain and then adds a shadow dentry under the parent directory in the local domain. SpanFS wraps the blocks to be modified in two JBD2 handles of the remote domain and local domain to achieve atomicity in the two domains respectively.

The consistency of a distributed object is equivalent to the integrity of the bidirectional index: (1) A shadow dentry is valid only if the remote dentry it points to exists and points back to it too. (2) A remote object (the remote dentry and inode) is valid only if its shadow dentry exists and points to it.

3.4.1 Stale object deletion.

The stale object deletion will validate any shadow dentry by checking whether its remote dentry exists when performing *lookup* and *readdir*. Specifically, for a distributed object SpanFS first locks the span directory via the mutex lock, then looks up the remote dentry under the span directory using the embedded index information and unlocks the span directory in the end. If the remote dentry does not exist, SpanFS deletes the shadow dentry. Note that the parent directory would be locked by the VFS during the above process.

3.4.2 Garbage collection (GC)

The GC mechanism deals with the scenario when the remote dentry and inode exist while the shadow dentry is lost. Under this circumstance, the file system consistency is not impaired since the remote object can never be seen by applications. But the remote dentry and inode will occupy storage space silently and should be collected.

During mounting if SpanFS finds out that it has just gone through a system crash, SpanFS will generate a background garbage collection thread to scan the span directories. The GC thread verifies the integrity of each remote dentry in each span directory at runtime silently, and removes the remote objects without shadow dentries.

Two-phase validation. In order to avoid contention with the normal operations, the GC thread performs two-phase validation: the scan phase and the integrity validation phase. During the scan phase the background GC thread locks the span directory via the mutex lock, reads the dentries under it and then unlocks the span directory. Then the GC thread validates each scanned remote dentry's integrity. To avoid locking the span directory for a long time, the GC thread reads a small number of remote dentries (4 KB by default) each time until all the remote dentries have been scanned and validated.

3.4.3 Avoiding deadlocks and conflicts

To avoid deadlocks and conflicts, for all operations that involve the creation or deletion of a distributed object we should first lock the parent directory of the shadow dentry and then lock the span directory of the remote dentry. By doing so, SpanFS can guarantee that new remote objects created by the normal operations will not be removed by the background GC thread by mistake and can avoid any deadlocks.

During the integrity validation phase, the GC thread first locks the parent directory (read from the bidirectional index on the remote dentry) and then looks up the shadow dentry under it. If the shadow dentry is found and points to the remote dentry, the GC thread does nothing and unlocks the parent directory. Otherwise, the GC thread locks the span directory and then tries to delete the remote object. If the remote object does not exist, it might be deleted by the normal operation before the integrity validation phase. For such case, the GC thread does nothing. The GC thread unlocks the span directory and the parent directory in the end.

For normal operations such as *create()* and *unlink()*, SpanFS first locks the span directory of the remote dentry, then creates or deletes the remote object and the shadow dentry, and unlocks the span directory in the end. Note that the parent directory would be locked by the VFS during the above process. For *unlink()* the inode will not be deleted until its link count drops to zero.

3.4.4 Parallel two-phase synchronization

The VFS invokes *fsync()* to flush the dirty data and corresponding metadata of a target object to disk. As the dentry and its corresponding inode may be scattered on two domains, SpanFS should persist the target object and its ancestor directory objects, their shadow/remote dentries if distributed, along the file path.

In order to reduce the distributed *fsync()* latency, we propose a parallel two-phase synchronization mechanism: the committing phase and the validating phase. During the committing phase SpanFS traverses the target object and its ancestor directories except for the SpanFS root directory. For each traversed object, SpanFS wakes up the journaling thread in its parent directory's domain to commit the running transaction and then records the committing transaction id in its VFS inode's field. Note that if there does not exist a running transaction, SpanFS does nothing. This situation may occur when the running transaction has been committed to disk by other synchronization actions such as periodic transaction commits in JBD2. Then, SpanFS starts to commit the target object. In the end SpanFS traverses the target object and its ancestor directories again to validate whether the recorded transaction commits have completed. If some of the commits have not completed, SpanFS should wait on the wait queues for their completion.

The synchronization mechanism utilizes JBD2 client-server transaction commit architecture. In JBD2, the client thread wakes up the journaling thread to commit the running transaction and then waits for its completion. In SpanFS, we leverage the journaling thread in each domain to commit the running transactions in parallel.

In order to avoid redundant transaction commits, we use flags (*ENTRY_NEW*, *ENTRY_COMMIT* and *ENTRY_PERSISTENT*) for each object to record its state. Ext4 with no journaling has the counterpart of our committing phase but does not have validating phase, which could potentially lead to inconsistencies.

During the committing phase SpanFS will clear the *ENTRY_NEW* flag of each traversed object. If cleared, SpanFS stops the committing phase. The *ENTRY_COMMIT* flag of the object would be set after the transaction has been committed. If not set, SpanFS would commit the transaction in its parent directory's domain and wait for the completion during the validating phase. During the validating phase, SpanFS will set the *ENTRY_PERSISTENT* flags of the traversed objects when all the recorded transaction commits have been completed. If set, SpanFS stops the validating phase.

3.4.5 Rename

The rename operation in Linux file systems tries to move a source object under the source directory to the destination object with the new name under the destination directory. SpanFS achieves atomicity of the rename operation through the proposed *ordered transaction commit mechanism*, which controls the commit sequence of the JBD2 handles on multiple domains for the rename operation. SpanFS ensures the commit order by marking each handle with *h_sync* flag, which would force JBD2 to commit the corresponding running transaction and wait for its completion when the handle is stopped.

For the case that the destination object does not exist, three steps are needed to complete a rename operation. Due to space limitation, we only demonstrate the case where the source object is a distributed object. The shadow dentry of the source object resides in Domain A. The inode of the source directory that contains the shadow dentry of the source object also resides in Domain A. The remote dentry and the inode of the source object resides in Domain B. The inode of the destination directory resides in Domain C. SpanFS starts a JBD2 handle for each step.

Step 1: SpanFS adds a new shadow dentry, which points to the remote dentry of the source object, to the destination directory in Domain C. If system crashes after this handle reaches the disk, the bidirectional index between the old shadow dentry and the remote dentry of the source object is still complete. The newly added dentry will be identified as stale under the destination directory and be removed at next mount.

Step 2: The remote dentry of the source object is altered to point to the newly added shadow dentry in Step 1. Then the bidirectional index between the old shadow dentry and the remote dentry of the source object becomes unidirectional while the bidirectional index between the new shadow dentry and the remote dentry is built. As long as the handle reaches disk, the old shadow dentry of the source object in Domain A is turned stale and the rename operation is essentially done.

Step 3: Remove the old shadow dentry of the source object under the source directory.

During the above process, JBD2 handles could be merged if they operate on the same domain. If the step needs to lock the span directory, it must start a new handle to avoid deadlocks, esp. step 2.

For the case that the destination object already exists, SpanFS first has the existing shadow dentry of the destination object tagged with *TAG_COMMON*, then adds another new shadow dentry tagged with both *TAG_NEWENT* and *TAG_COMMON* in the destination directory in Step 1. Moreover, two extra steps are needed to complete the rename: Step 4 to remove the inode and the remote dentry of the destination object and step 5 to delete the existing shadow dentry of the destination object and untag the newly added shadow dentry under the destination directory. As the existing shadow dentry has the same name as the newly added dentry under the destination directory, we use the tags to resolve conflicts in case of system crashes. Specifically, a dentry tagged with *TAG_COMMON* should be checked during *lookup()*. If there exist two tagged dentries with the same name under a directory, SpanFS will remove the one without integral bidirectional index. If their bidirectional indices are both integral, the one with *TAG_NEWENT* takes precedence and the other is judged as stale and should be removed.

3.5 Discussion

The approach introduced in this paper is not the only way to scale file systems. Another way to providing parallel file system services is running multiple file system instances by using disk partitions (virtual block devices in [26]), stacking a unified namespace on top of them and maintaining crash consistency across them, which is similar to Unionfs [42] in the architecture. We previously adopt this approach as an extension to MultiLanes [27] to reduce contention inside each container [26]. However, this approach has several drawbacks: First, managing a number of file systems would induce administration costs [31]. Second, adjusting storage space among multiple file systems and partitions on demand also introduces management cost. Although we can leverage the virtual block device of MultiLanes to support storage space overcommitment, it comes with a little cost of lightweight virtualization [27]. Third, the cost of namespace unification will increase with the increasing number of partitions [42].

4 Evaluation

We evaluate the performance and scalability of SpanFS against Ext4 using a set of micro and application-level benchmarks.

4.1 Test Setup

All experiments were carried out on an Intel 32-core machine with four Intel(R) Xeon(R) E5-4650 processors (with the hyperthreading capability disabled) and 512 GB memory. Each processor has eight physical cores running at 2.70 GHZ. All the experiments are carried out on a Fusion-IO SSD (785 GB MLC Fusion-IO ioDrive). The experimental machine runs a Linux 3.18.0 kernel. We compile a separated kernel with *lock stat* enabled to collect the lock contention statistics.

We use 256 GB of the SSD for evaluation. We evaluate SpanFS with 16 domains and 4 domains in turn. We statically allocate 16 GB storage space to each domain for the 16 domain configuration and 64 GB storage space to each domain for the 4 domain configuration. For SpanFS with 16 domains, each domain has 64 MB journaling size, yielding a total journaling size of 1024 MB. To rule out the effects of different journaling sizes, the journaling sizes of both SpanFS with 4 domains and Ext4 are all set to 1024 MB, respectively. Both SpanFS and Ext4 are mounted in ordered journal mode unless otherwise specified.

Kernel Patch. The VFS uses a global lock to protect each super block's inode list, which can cause contention. We replace the super block's inode list with per-core lists and use per-core locks to protect them. We apply this patch to both the baseline and SpanFS.

4.2 Performance Results

4.2.1 Metadata-Intensive Performance

We create the micro-benchmark suite called *catd*, which consists of four benchmarks: *create*, *append*, *truncate* and *delete*. Each benchmark creates a number of threads performing the corresponding operation in parallel and we vary the number of threads from 1 to 32.

Create: Each thread creates 10000 files under its private directory.

Append: Each thread performs a 4 KB buffered write and a *fsync()* to each file under its private directory.

Truncate: Each thread truncates the appended 4 KB files to zero-size.

Delete: Each thread removes the 10000 truncated files.

We run the benchmark in the order of *create-append-truncate-delete* in a single thread and multiple threads concurrently. Figure 4 shows that SpanFS performs much better than Ext4 except for the *create* benchmark.

For the *create* benchmark SpanFS performs worse than Ext4 for two reasons: Ext4 has not encountered severe scalability bottlenecks under this workload, and SpanFS introduces considerable overhead as it needs to create two dentries for each distributed object. Ext4 is 113% and 42% faster than SpanFS with 16 domains at one core and at 32 cores, respectively. Note that the 4 domain configuration performs better than the 16 domain configuration mainly due to that the percentage of the distributed objects in SpanFS with 4 domains is lower.

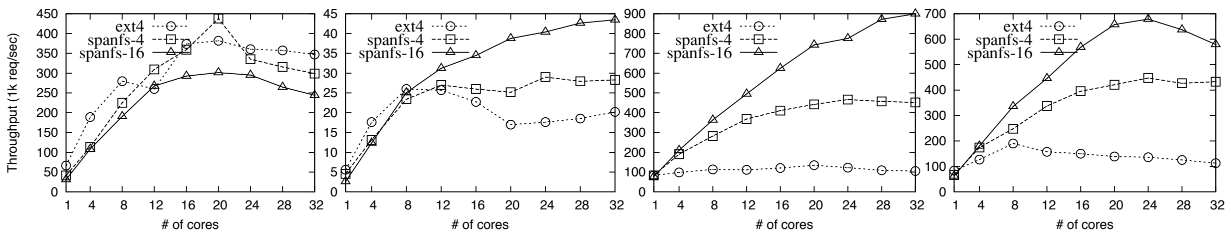
Ext4		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
sbi->s_orphan_lock	478 k	534 s (1117.32 µs)
journal->j_wait_done_commit	845 k	100.4 s (112.10 µs)
journal->j_checkpoint_mutex	71 k	56.5 s (789.70 µs)
journal->j_list_lock	694 k	10.5 s (14.64 µs)
journal->j_state_lock-R	319 k	9.8 s (28.58 µs)

SpanFS-16		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
journal->j_checkpoint_mutex	27 k	15.1 s (557.96 µs)
inode_hash_lock	323 k	8.1 s (25.07 µs)
sbi->s_orphan_lock	124 k	4.3 s (34.51 µs)
journal->j_wait_done_commit	287 k	3.4 s (11.07 µs)
ps->lock (Fusionio driver)	789 k	2.4 s (2.87 µs)

Table 3: **The top 5 hottest locks**

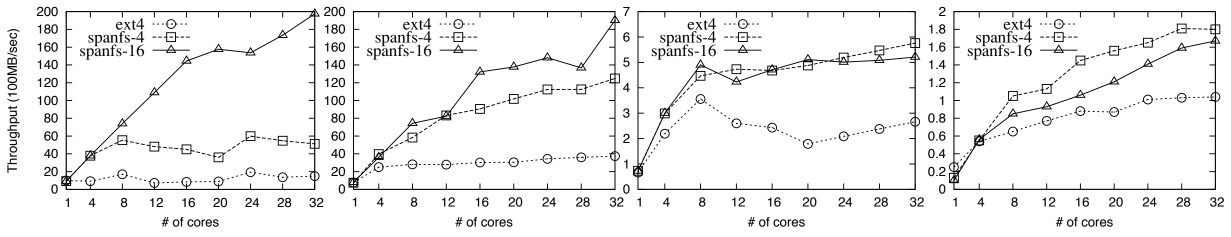
As shown in Figure 4, for the *append*, *truncate*, *delete* benchmark, SpanFS significantly outperforms Ext4 beyond a number of cores due to the reduced contention and better parallelism. As the *fsync()* in *append* may span several domains to persist the objects along the path and the *delete* benchmark involves the deletion of two dentries for each distributed object, there exists some overhead for these two benchmarks. Specifically, SpanFS is 113% and 33% slower than Ext4 for these two benchmarks at one single core. However, due to the reduced contention, SpanFS with 16 domains outperforms Ext4 by 1.15X, 7.53X and 4.13X at 32 cores on the *append*, *truncate* and *delete* benchmark, respectively.

To understand the performance gains yielded by SpanFS, we run the *catd* benchmark at 32 cores in a separated kernel with *lock stat* enabled. Table 3 shows that Ext4 spends substantial time acquiring the hottest locks during the benchmarking. In contrast, the total wait time



(a) create (b) append (c) truncate (d) delete

Figure 4: **Catd.** *This figure depicts the overall throughput (operations per second) with the benchmark create, append, truncate and delete, respectively.*



(a) sequential buffered write (b) random buffered write (c) direct write (d) synchronous write

Figure 5: **IOzone.** *This figure shows the total throughput of IOzone on SpanFS against Ext4 under sequential buffered writes, random buffered writes, sequential direct writes and sequential synchronous writes (open with O_SYNC), respectively.*

of the hot locks in SpanFS has been reduced by 20X.

4.2.2 Data-Intensive Performance

IOzone. The IOzone [3] benchmark creates a number of threads, each of which performs 4KB writes to a single file which ends up with 512 MB. Figure 5 shows that SpanFS scales much better than Ext4, leading to significant performance improvements. Specifically, SpanFS with 16 domains outperforms Ext4 by 1226%, 408%, 96% and 60% under the four I/O patterns at 32 cores, respectively. For direct I/O, Ext4 scales poorly due to the contention when logging the block allocation.

Sysbench. We run multiple single-threaded sysbench instances in parallel, each of which issues 4 KB writes. Each instance operates over 128 files with a total write traffic of 512 MB. Figure 6 shows that SpanFS scales well to 32 cores, bringing significant performance improvements. Specifically, SpanFS with 16 domains is 4.38X, 5.19X, 1.21X and 1.28X faster than Ext4 in the four I/O patterns at 32 cores, respectively.

4.2.3 Application-Level Performance

Filebench. We use Filebench [6] to generate application-level I/O workloads: the Fileserver and Varmail workloads. The Varmail workload adopts the parameter of 1000 files, 1000000 average directory width, 16 KB average file size, 1 MB I/O size and 16 KB average append size. The Fileserver workload adopts the parameter of 10000 files, 20 average directory width, 128 KB average file size, 1 MB I/O size and 16 KB average append size. We run multiple single-threaded Filebench instances in parallel and vary the number of instances from 1 to 32. Each workload runs for 60 s.

As shown in Figure 7(a) and Figure 7(b) , for the Fileserver and Varmail workloads SpanFS in all the two configurations scales much better than Ext4. SpanFS with 16 domains outperforms Ext4 by 51% and 73% under the Fileserver and Varmail workloads at 32 cores, respectively. We have also evaluated the performance of SpanFS against Ext4 in data journal mode under the Varmail workload. Figure 7(c) shows that SpanFS with 16 domains outperforms Ext4 by 88.7% at 32 cores.

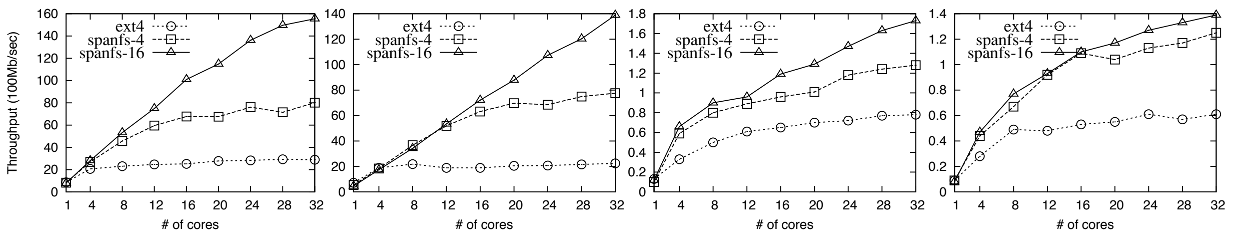
Dbench. We use Dbench [4] to generate I/O workloads that mainly consist of *creates, renames, deletes, stats, finds, writes, getdents* and *flushes*. We choose Dbench to evaluate SpanFS as it allows us to illuminate performance impact of the rename operation overhead on a realistic workload. We run multiple single-threaded Dbench instances in parallel.

Due to the overhead, SpanFS with 16 domains is 55% slower than Ext4 at one single core. However, as shown in Figure 7(d), due to the reduced contention and better parallelism SpanFS with 16 domains outperforms Ext4 by 16% at 32 cores.

4.2.4 Comparison with other file systems

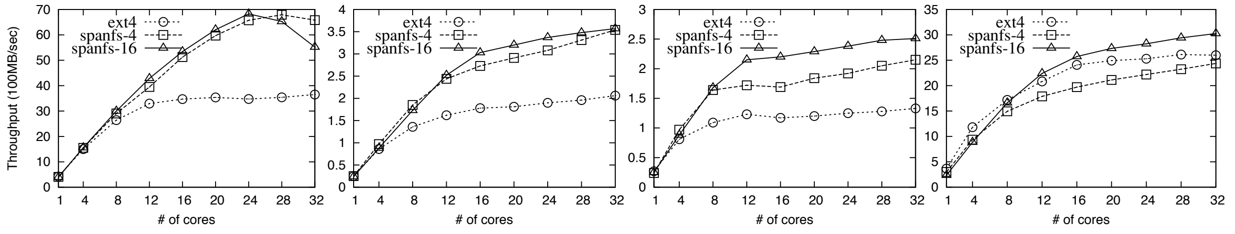
We make a comparison of SpanFS with other file systems on scalability using the Fileserver workload. Figure 8 (a) shows that SpanFS with 16 domains achieves much better scalability than XFS, Btrfs and ZFS.

We also make a comparison with MultiLanes+ [26], an extended version of our previous work [27]. As MultiLanes+ stacks a unified namespace on top of multiple virtual block devices, it comes at the cost of namespace unification. We evaluate SpanFS with 32 domains against MultiLanes+ with 32 disk partitions using the *create*



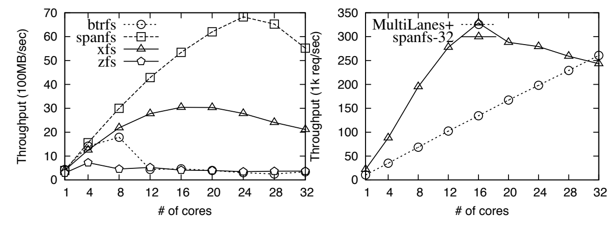
(a) sequential buffered write (b) random buffered write (c) synchronous write (d) random synchronous write

Figure 6: **Sysbench.** *This figure depicts the total throughput of sysbench on SpanFS against Ext4 under sequential buffered writes, random buffered writes, sequential synchronous writes and random synchronous writes, respectively. The first two buffered I/O patterns do not issue any* fsync() *while the synchronous I/O patterns issue a* fsync() *after each write.*



(a) fileserver (b) varmail (c) varmail (*data journal*) (d) dbench

Figure 7: **Filebench and Dbench.** *This figure depicts the total throughput of Filebench (Fileserver and Varmail) and Dbench on SpanFS against Ext4, respectively.*



(a) fileserver (b) create

Figure 8: **Comparison with other major file systems.** *The throughput of Filebench Fileserver on SpanFS against other three major file systems: XFS, Btrfs and ZFS and the throughput of the* create *benchmark on SpanFS against MultiLanes+.*

benchmark. The journaling size of each domain/partition is set to 128 MB. Figure 8 (b) shows the lines generated by SpanFS against MultiLanes+. As the *create* operation need to perform namespace unification which is expensive in MultiLanes+, SpanFS performs much better than MultiLanes+ from 1 to 28 cores. Especially, SpanFS is faster than MultiLanes+ between 72% and 185% from 1 core to 20 cores. Due to the increased contention, the performance improvement shrinks from 24 cores to 32 cores.

4.2.5 Garbage Collection Performance

We evaluate the time the GC takes to scan different numbers of files. We use the *create* benchmark to prepare a set of files in parallel using 32 threads under 32 directories and then remount the file system with the GC thread running at the background. As the GC thread only needs to run when SpanFS finds out that it has just gone through a crash we manually enable the GC thread. Table 4 shows that the GC thread only needs 2.4 seconds to scan and validate all the remote objects when there are 320000 files in the file system, and the time only increases to 20 seconds when there exist 3.2 millions of files. The cost the GC thread incurred is relatively small thanks to the high performance provided by the SSD.

# of files	32000	320000	3200000
# of remote dentries	30032	300030	3000030
Time	1071 ms	2403 ms	20725 ms

Table 4: **Garbage collection performance.** *The time taken to scan the span directories to perform garbage collection. As there exist normal objects, the number of remote dentries represents the actual number of dentries that the GC thread has scanned and validated.*

Then we measure the overhead that the background GC activities contribute to the foreground I/O workloads. Specifically, we prepare 3.2 millions of files as the above does, remount the file system with the GC thread running and then immediately run 32 Varmail instances in parallel. The Varmail workload runs for 60 s.

We measure the aggregative throughput of the 32 Varmail instances. Compared with the normal case without the GC thread running, the total throughput of the Varmail workload has been degraded by 12% (357 MB vs 313 MB), and the GC thread has taken 21950 ms to validate 3024296 remote objects. The number of the validated remote objects is higher than the number in Table 4 as the GC has scanned the remote objects created by the running Varmail workload. Meanwhile, during the above process, the GC thread has found four false invalid remote objects. These false invalid objects are created by the Varmail workload and are deleted before the GC integrity validation phase. This test also demonstrates that SpanFS can correctly deal with the conflicts between the GC thread and the normal I/O operations.

	Ext4	SpanFS-16
Open without VFS cache	13.9 μs	24.7 μs
Open with VFS cache	3.4 μs	3.5 μs
Rename (1 core)	24 μs	609 μs
Rename (32 cores)	65 μs	2591 μs
unmount	4.323 s	5.272 s
mount	0.021 s	0.086 s

Table 5: **The operation latency.**

4.2.6 Overhead Analysis

We use the average operation latency reported by the above Dbench running in Section 4.2.3 to show the rename overhead. As shown in Table 5, SpanFS with 16 domains is 24X and 39X slower than Ext4 at one core and at 32 cores for the rename operation in Dbench, respectively. We also create a micro-benchmark to evaluate the rename overhead. The rename benchmark renames 10000 files to new empty locations and renames 10000 files to overwrite 10000 existing files. The result shows that SpanFS is 25X and 84X slower than Ext4 for the rename and overwritten rename.

We construct a benchmark *open* to evaluate the overhead of validating the distributed object's integrity during *lookup()* in SpanFS. The benchmark creates 10000 files, remounts the file system to clean the cache and then opens the 10000 files successively. We measure the average latency of each operation. As shown in Table 5, the average latency in Ext4 is around 13.9 μs. In contrast, the average latency is around 24.7 μs in SpanFS with 16 domains. We then open the 10000 files again without remounting the file system. The results show that with the VFS cache SpanFS exhibits almost the same performance with Ext4.

We create a benchmark *mount* to evaluate the performance of the *mount* and *unmount* operation in SpanFS. The *mount* benchmark untars the compressed Linux 3.18.0 kernel source, then unmounts the file system and mounts it again. Table 5 shows the time taken for SpanFS and Ext4 to unmount and mount the file system. As SpanFS builds the domains in sequence, SpanFS with 16 domains performs significantly worse than Ext4 for the mount operation. Nevertheless, the time taken to mount SpanFS only costs 86 ms.

5 Related Work

Scalable I/O stacks. Zheng et al. [43] mainly focus on addressing the scalability issues within the page cache layer, and try to sidestep the kernel file system bottlenecks by creating one I/O service thread for each SSD. However, their approach comes at the cost of communication between application threads and the I/O threads.

Some work that tries to scale in-memory file systems has emerged. ScaleFS [21] uses per core operation logs to achieve scalability of in-memory file system operations that can commute [19]. Hare [25, 24] tries to build a scalable in-memory file system for multi-kernel OS. However, these work does not focus on the scalability issues of the on-disk file systems that need to provide durability and crash consistency.

Wang et al. [40] leverage emerging non-volatile memories (NVMs) to build scalable logging for databases, which uses a global sequence number (GSN) for dependency tracking between update records and transactions across multiple logs. However, due to the need of complex dependency tracking, applying their approach to the file systems needs to copy the updates to the journaling layer, which will introduce copying overhead that has almost been eliminated in the file system journaling [39]. Meanwhile, their work needs the support of emerging NVMs.

Isolated I/O stacks. Some work that tries to build isolated I/O stacks shares some similarities with the domain abstraction in our work in functionality. MultiLanes [27] builds an isolated I/O stack for each OS-level container to eliminate contention. Vanguard [36] and its relative Jericho [32] build isolated I/O stacks called slices and place independent workloads among the slices to eliminate performance interference by assigning the top-level directories under the root directory to the slices in a round-robin manner. IceFS [31] partitions the on-disk resources among containers called cubes to provide isolated I/O stacks mainly for fault tolerant and provides a dedicated running transaction for each container to enable parallel transaction commits. However, these work cannot reduce the contention within each single workload that runs multiple threads/processes as it is hosted inside one single isolated I/O stack.

In contrast with the above work, our work distributes all files and directories among the domains to achieve scalability and proposes a set of techniques to build a global namespace and to provide crash consistency.

Although the domain abstraction in our work shares some similarities with the cube abstraction of IceFS [31], they differ in the following aspects. First, the cubes of IceFS still share the same journaling instance, which can cause contention when multiple cubes allocate log space for new transactions simultaneously. Meanwhile, their approach may still need to serialize parallel checkpoints to make free space due to the single log shared by multiple cubes. In contrast, each domain in SpanFS has its own journaling instance. Second, IceFS does not focus on the lock contention within the block buffer cache layer while SpanFS provides a dedicated buffer cache address space for each domain to avoid such contention. Third, although IceFS supports dynamic block group allocation, their paper does not describe how to provide crash consistency during allocation. In contrast, our work provides a detailed design of the block group reallocation mechanism as well as how to maintain crash consistency during reallocation.

The online adjusting of each domain's size in the unit of block groups shares some similarities with the Ext2/3 online resizing [20]. However, the Ext2/3 online resizing only focuses on adjusting one file system's size. Our work provides a design on online adjusting of the storage space among multiple domains on demand and maintaining crash consistency during reallocation.

RadixVM [18] implements a scalable virtual memory address space for non-overlapping operations in their research OS. However, applying their approach to the buffer cache address space needs to modify the Linux kernel. In contrast, we leverage the Linux OS block architecture to provide a dedicated buffer cache address space for each domain to avoid the lock contention.

Scalable kernels. Disco [12] and Cerberus [37] run multiple operating systems through virtualization to provide scalability. Cerberus [37] provides a consistent clustered file system view on top of the virtual machines (VMs). However, their approach comes with the cost of inter-VM communication. Moreover, their paper does not explicitly discuss how to maintain consistency of the clustered file system in case of system crashes. Hive [14] and Barrelfish [8] achieve scalability on many-core through the multikernel model. Some work proposes new OS structures to achieve scalability on many-core, such as Corey [10], K42 [28] and Tornado [22]. SpanFS is influenced and inspired by these work but focuses on scaling file systems on fast storage as well as providing crash consistency.

File system consistency check. NoFS [17] stores the backpointers in data blocks, files and directories to verify the file system inconsistencies online, avoiding the journaling overhead. However, as NoFS cannot verify the inconsistencies of allocation structures such as inode bitmap, it needs to scan all the blocks and inodes to build the allocation information at mount time [17]. In contrast, SpanFS only needs to perform GC when it has gone through a crash and only needs to scan the remote dentries under the span directories rather than the whole device in case of a system crash.

Distributed file system. Some distributed file systems, such as Ceph [41] and IndexFS [34], partition the global namespace across computer nodes to provide parallel metadata service. These work relies on the interconnected network in a cluster to maintain a consistent view across machines. In contrast, SpanFS relies on the CPU cache coherence prototype to maintain consistency on data structures within a single many-core machine.

6 Conclusion and Future Work

In this paper, we first make an exhaustive analysis of the scalability bottlenecks of existing file systems, and attribute the scalability issues to their centralized design, especially the contention on shared in-memory data structures and the serialization of internal actions on devices. Then we propose a novel file system SpanFS to achieve scalability on many cores. Experiments show that SpanFS scales much better than Ext4, bringing significant performance improvements.

In our future work, we will implement the online adjusting of each domain's size, explore the adjusting policies and evaluate their performance. In our current prototype, the number of domains is fixed. We will explore the dynamic domain creation strategy.

7 Acknowledgements

We would like to thank our shepherd Chia-Lin Yang and the anonymous reviewers for their valuable suggestions that help improve this paper significantly. Junbin Kang and Benlong Zhang have contributed equally to this work. Junbin Kang and Tianyu Wo are the corresponding authors of this paper. This work is supported in part by China 973 Program (2011CB302602), National Natural Science Foundation of China (91118008, 61322207), China 863 program (2013AA01A213), HGJ Program (2013ZX01039002-001) and Beijing Higher Education Young Elite Teacher Project.

References

[1] SysBench, https://github.com/akopytov/sysbench.

[2] OpenZFS, http://open-zfs.org/wiki/Main_Page.

[3] IOzone Benchmark, http://www.iozone.org/.

[4] Dbench, https://dbench.samba.org/.

[5] Ext4 Disk Layout. https://ext4.wiki.kernel.org /index.php/Ext4_Disk_Layout. Accessed May 2015.

[6] Filebench. http://filebench.sourceforge.net/wiki /index.php/Main_Page.

[7] lockstat. https://www.kernel.org/doc/Documentation/ locking/lockstat.txt.

[8] BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T. L., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (2009).

[9] BJØRLING, M., AXBOE, J., NELLANS, D. W., AND BONNET, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *SYSTOR* (2013).

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *OSDI* (2008).

[11] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *OSDI* (2010).

[12] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. DISCO: running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[13] Cao, M., Bhattacharya, S., and Ts'o, T. Ext4: The next generation of ext2/3 filesystem. In *2007 Linux Storage & Filesystem Workshop, LSF 2007* (2007).

[14] Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. Hive: Fault containment for shared-memory multiprocessors. In *SOSP* (1995).

[15] Chen, F., Lee, R., and Zhang, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA* (2011).

[16] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic crash consistency. In *SOSP* (2013).

[17] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency without ordering. In *FAST* (2012).

[18] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys* (2013).

[19] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP* (2013).

[20] Dilger, A. E. Online ext2 and ext3 filesystem resizing. In *Ottawa Linux Symposium 2002* (2002).

[21] Eqbal, R. ScaleFS: A multicore-scalable file system. Master's thesis, Massachusetts Institute of Technology, Aug. 2014.

[22] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI* (1999).

[23] Hagmann, R. B. Reimplementing the cedar file system using logging and group commit. In *SOSP* (1987).

[24] III, C. G. *Providing a Shared File System in the Hare POSIX Multikernel*. PhD thesis, Massachusetts Institute of Technology, June 2014.

[25] III, C. G., Sironi, F., Kaashoek, M. F., and Zeldovich, N. Hare: a file system for non-cache-coherent multicores. In *EuroSys* (2015).

[26] Kang, J., Hu, C., Wo, T., Zhai, Y., Zhang, B., and Huai, J. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. An extended verison of [27] submitted to a journal.

[27] Kang, J., Zhang, B., Wo, T., Hu, C., and Huai, J. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *FAST* (2014).

[28] Krieger, O., Auslander, M. A., Rosenburg, B. S., Wisniewski, R. W., Xenidis, J., Silva, D. D., Ostrowski, M., Appavoo, J., Butrico, M. A., Mergen, M. F., Waterland, A., and Uhlig, V. K42: building a complete operating system. In *EuroSys* (2006).

[29] Lee, S., Moon, B., and Park, C. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD* (2009).

[30] Liu, R., Zhang, H., and Chen, H. Scalable read-mostly synchronization using passive reader-writer locks. In *USENIX ATC* (2014).

[31] Lu, L., Zhang, Y., Do, T., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Physical disentanglement in a container-based file system. In *OSDI* (2014).

[32] Mavridis, S., Sfakianakis, Y., Papagiannis, A., Marazakis, M., and Bilas, A. Jericho: Achieving scalability through optimal data placement on multicore systems. In *IEEE MSST* (2014).

[33] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (2001).

[34] Ren, K., Zheng, Q., Patil, S., and Gibson, G. A. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC* (2014).

[35] Rodeh, O., Bacik, J., and Mason, C. BTRFS: the Linux B-Tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 9:1–9:32.

[36] Sfakianakis, Y., Mavridis, S., Papagiannis, A., Papageorgiou, S., Fountoulakis, M., Marazakis, M., and Bilas, A. Vanguard: Increasing server efficiency via workload isolation in the storage I/O path. In *Proceedings of the ACM Symposium on Cloud Computing* (2014).

[37] Song, X., Chen, H., Chen, R., Wang, Y., and Zang, B. A case for scaling applications to many-core with OS clustering. In *EuroSys* (2011).

[38] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *USENIX ATC* (1996).

[39] Tweedie, S. C. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[40] Wang, T., and Johnson, R. Scalable logging through emerging non-volatile memory. *PVLDB 7*, 10 (2014), 865–876.

[41] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *OSDI* (2006).

[42] Wright, C. P., Dave, J., Gupta, P., Krishnan, H., Quigley, D. P., Zadok, E., and Zubair, M. N. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage (TOS) 2*, 1 (2006), 74–105.

[43] Zheng, D., Burns, R. C., and Szalay, A. S. Toward millions of file system IOPS on low-cost, commodity hardware. In *SC* (2013).