MAX: A Multicore-Accelerated File System for Flash Storage

Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu, *Department of Computer Science and Technology, Tsinghua University, and Beijing National Research Center for Information Science and Technology* (*BNRist*)

https://www.usenix.org/conference/atc21/presentation/liao

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

MAX: A Multicore-Accelerated File System for Flash Storage

Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

The bandwidth of flash storage has been surging in recent years. Employing multicores to fully unleash its abundant bandwidth becomes a necessary step towards building high performance storage systems. This paper presents the design and implementation of MAX, a multicore-friendly log-structured file system (LFS) for flash storage. With three main techniques, MAX systematically improves the scalability of LFS while retaining the flash-friendly design. First, we propose a new reader-writer semaphore to scale the user I/Os with negligible impact on the internal operations of LFS. Second, we introduce file cell to scale the access to in-memory index and cache while delivering concurrency- and flash-friendly on-disk layout. Third, to fully exploit the flash parallelism, we advance the single log design with runtime-independent log partitions, and delay the ordering and consistency guarantees to crash recovery. We implement MAX based on the F2FS in the Linux kernel. Evaluations show that MAX significantly improves scalability, and achieves an order of magnitude higher throughput than existing Linux file systems.

1 Introduction

The bandwidth of solid-state drives (SSDs) has been quickly increasing over the past decade [23, 24, 55]. To unleash full throughput potentials from such improvement, efficiently utilizing multicores to handle concurrent I/Os becomes a must. Currently, Non-Volatile Memory Express (NVMe) protocol [9] and multi-queue block layer [13] have already laid a multicore-friendly foundation at the driver layer. Additionally, in the upper software stack, great efforts have been made to increase the scalability [29, 38, 40, 43, 50, 56].

Nonetheless, an important question is still left unanswered: whether the log-structured file systems (LFS) atop the flash-based SSDs adapt well to the scaling of cores. LFS, initially introduced in Sprite LFS [52], builds on a simple idea: organizing the address space as an append-only log. This design essentially converts random writes into sequential ones, which not only aligns with the I/O preference of legacy hard disk drive (HDD), but also is a common practice of file system for flash storage [35, 37, 49]. First, due to the intrinsic NAND idiosyncrasies, the sequential performance of most flash SSDs is still significantly higher than the random performance [24–28, 53, 54]. Second, the zoned namespace (ZNS), an optimized interface for flash SSDs, is available in NVMe spec and under increasing promotion [4, 6, 12, 58]. ZNS favors log-structured writes, and existing LFSes (e.g., F2FS [37]) can directly run atop it. Therefore, LFS is a promising architecture for flash SSD, and understanding its performance in the multicore context yields great significance.

Hence, we start this paper with a study on file systems (esp. LFS) throughput under concurrent and independent I/Os (§2). By increasing the number of CPU cores, we observe that most file systems scale relatively well on traditional storage devices (i.e., HDD and SATA SSDs). Surprisingly, the performance of file systems atop the modern NVMe SSDs suffers greatly from scaling. Most notably in F2FS, an LFS optimized for NVMe SSD, the performance peaks at only 18 cores, and a further scaling to 72 cores causes throughput drop by nearly 30%. The I/O utilization of F2FS on NVMe SSD is only 20%.

Through profiling, we conclude that the unscalable data structures of the file system cost a considerable amount of CPU cycles and thus bottleneck the performance. The root cause of the inefficiency comes from a legacy choice: using shared data structures to aggregate file operations and I/Os for high performance. Such philosophy essentially trades CPU cycles for high device I/O utilization. However, for NVMe SSD, this trade-off breaks as the CPU cycles are no longer negligible for high performance drives.

While many research have been conducted to understand and improve such inefficiency [29, 48, 56], they mostly focus on journaling file systems (e.g., Ext4 [44]) and therefore can not directly be applied to LFS due to different designs. For instance, LFS uses a checkpoint mechanism instead of journaling for persistence and consistency, thereby unable to use techniques such as parallel journaling for scalability [29, 56]. Hence, we first analyze the root causes inside the LFS.

Here, we decompose the LFS internals from top to bottom into three levels, the Concurrency Control (CC) level, the In-Memory Data Structure (IMDS) level and Space Allocation (SA) level, as shown in Table 1. We find that the lock con-

*Jiwu Shu (shujw@tsinghua.edu.cn) is the corresponding author.

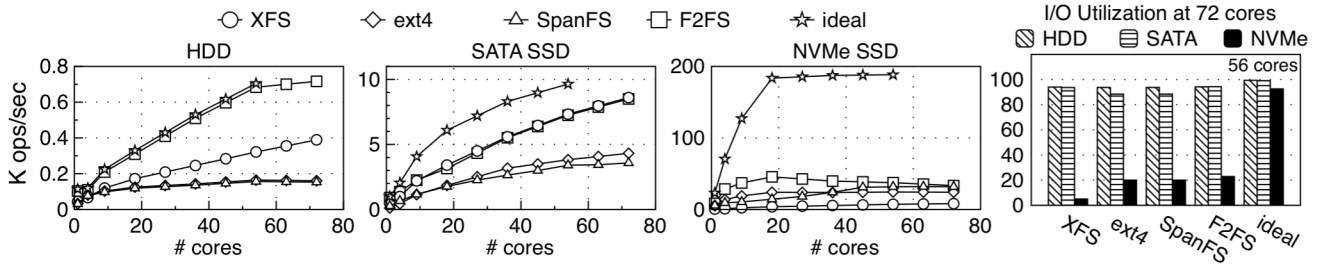

Figure 1: **Scalability problem evaluation.** *We make two major observations: (1) On slow hard disk drive and SATA SSD, most file systems scale well, and the storage device is almost saturated. (2) In contrast, on high performance NVMe SSD, the performance of almost all file systems starts to drop after 18 cores, and the device is underutilized. HDD: Seagate ST1000NX0313; SATA SSD: Samsung 850 Pro; NVMe SSD: Intel DC P3700. Described in §2.*

Lock level \ Lock type	Shared mode	Exclusive mode
Concurrency Control (CC)	write operations (e.g., `write`)	global operations (e.g.,`sync`)
In-Memory Data Structure (IMDS)	index read operations (e.g., `write`)	index write operations (e.g., `create`)
Lock level \ Lock type	**Mutual exclusion lock**	
space allocation (SA)	durability operations (e.g.,`fsync`)	

Table 1: **Current practices of the file system sharing.**

tentions caused by the shared data structures in each level serialize independent I/Os, and further prevent applications from taking full advantage of multicore-friendly design of NVMe and the abundant bandwidth of flash storage.

In this paper, we introduce the following three design principles to reduce the lock contentions of each level, and further scale the multicore performance of LFS for flash storage (§3).

- In CC level, parallelizing the external I/O requests (e.g., read and write syscalls) while keeping the internal operations (e.g., LFS checkpoint) efficient.
- In IMDS level, scaling the IMDS access while delivering flash- and concurrency-friendly on-disk format for concurrent persistence functions (e.g., `fsync`).
- In SA level, paralleling persistence functions while delegating (some) ordering and consistency to crash recovery.

We implement these ideas in Max (§4), a multicore accelerated file system for high performance flash storage, with a set of modifications of F2FS. Max replaces a global reader-writer semaphore (rwsem) of F2FS with a tailored rwsem (§4.1), reorganizes the IMDS (§4.2) and on-disk layout (§4.3) and slightly changes the crash recovery procedure (§4.4). During the development of Max, we surprisingly find some optimizations of F2FS restrict the multicore performance. We describe our solutions to maintain these optimizations while offering better scalability (§4.5). In a nutshell, Max restructures F2FS, enabling independent I/Os to concurrently enter the file system, concurrently access the IMDS and concurrently reach the persistent storage. We further study Max's performance (§5). Under a wide variety of micro- and macro-benchmarks, we show that Max achieves up to an order of magnitude higher throughput than existing Linux file systems. Further, evaluation against memory-backed tmpfs [51] indicates that, for certain file operations (e.g., appending blocks to private files), the performance of Max almost reaches the upper bound of VFS (virtual file system).

In summary, we provide three major contributions.

- We perform a study on the multicore scalability of LFS and demonstrate that the lock contentions from different levels as the major culprits.
- We propose Max removes unnecessary sharing of different levels by using a novel reader-writer semaphore, a new in-memory data structure abstraction and log partitions.
- Our evaluations show that Max outperforms existing Linux file systems by up to an order magnitude. For some file operations, Max comes close to the upper bound of VFS.

2 Background and Motivation

2.1 Understanding the Performance

We start with measuring the throughput of 5 file system setups (XFS [18], Ext4 [44], SpanFS [29], F2FS [37] and ideal) on three types of storage devices (HDD, SATA SSD and NVMe SSD). In the ideal setup, we partition the drive and run an independent F2FS on each partition. This enables each parallel process to execute in its dedicated file system without software level contention. As `fdisk` [2] only allows at most 56 partitions, we do not include performances of ideal setup beyond that. §5 further describes other details of the testbed.

In the experiment, we attach each core with one process and scale the number of cores from 1 to 72 (i.e., X axis in Figure 1). Each process runs for 60 seconds, and executes the following operations. First, the process creates a file in its own directory. Then the process issues 4 KB writes, invokes the `fsync` on each file and then deletes the file.

Figure 1 plots the results. First, we observe that existing file systems scale poorly on NVMe SSDs. For HDD and slower SATA SSD, the performance of many file systems (e.g., F2FS and XFS) is close to that of the ideal setup. For NVMe SSDs, the throughput of existing file systems is far from ideal. Instead of benefiting from the scalability, most file

Operations	Lock/Sharing	Overhead	Lock level	Description
write()	sbi->cp_rwsem	50.98%	CC	Mutual exclusion between checkpoint and write operations.
create()	nm_i->nat_tree_lock	3.74%	IMDS	Protecting a radix tree indexing of inode table.
	sbi->inode_lock	3.22%	IMDS	Protecting a list indexing of dirty inode.
	curseg->curseg_mutex	1.84%	SA	Serializing log-structured space allocation.
	nm_i->nid_list_lock	1.09%	IMDS	Protecting a central ever-increasing inode ID allocator.
unlink()	nm_i->nat_tree_lock	22.68%	IMDS	Protecting a radix tree indexing of inode table.
	im->ino_lock	6.54%	IMDS	Protecting a list and radix tree indexing of cached inode.
	sbi->inode_lock	3.05%	IMDS	Protecting a list indexing of dirty inode.
	node_inode	2.66%	IMDS	A user-invisible inode structure to trace all inode pages.
	sit_i->sentry_lock	2.41%	SA	Synchronizing concurrent access to segment info table.
fsync()	sbi->writepages	45.76%	SA	Enforcing the sequential log access.
	sit_i->sentry_lock	1.07%	SA	Synchronizing concurrent access to segment info table.

Table 2: **The scalability bottlenecks of F2FS.** *write(): overwriting blocks of private files, create()/unlink(): creating/deleting files in private directories, fsync(): invoking* `fsync` *on private files. Described in §2.2.*

system actually suffer from the increasing number of cores. For example, F2FS peaks at 18 cores, then starts to decline and ends up with a 30% performance loss. Further, in the rightmost plot of Figure 1, we observe that, even at the scale of 72 cores, most file systems do not efficiently utilize the I/O bandwidth of NVMe SSD.

The results from this experiment suggest that the performance of existing file systems are no longer bounded by the underlying device or the drive layer. Instead, the file system itself becomes the bottleneck and can not efficiently exploit the bandwidth of high performance drives.

2.2 Identifying Root Causes

Next, we investigate the CPU overhead distribution to identify the root causes of poor scalability in F2FS atop the NVMe SSDs. We use Linux performance analysis tools `perf` [8] to measure the overhead of each function call in terms of CPU cycles. We focus on four representative system calls (i.e., `write`, `create`, `unlink` and `fsync`) in a 72-core-scaling setup of F2FS. We observe that lock contention caused by unscalable data structure organization is a major source of overhead. Hence, we single out expensive lock operations, and identify their levels. Table 2 shows the overall results.

Lock cache coherence at CC level. The operations of LFS can be broadly classified into three categories: user read operations (e.g., `read` and `stat`), user write operations (e.g., `write` and `create`) and LFS internal operations (e.g., checkpoint). Most Linux file systems (FS) control the concurrency among user read operations and write operations by a relatively simple way: using a file-level inode reader-writer semaphore [1]. The concurrency control among independent writes and the global checkpoint is more complicated; as shown in Table 1, LFS usually employs a traditional reader-writer lock (e.g., `cp_rwsem` of F2FS and `ns_segctor_sem` of NILFS2 [35]) at CC level to grant access for writes and checkpoint. Independent writes can concurrently update disjoint parts of the FS image, and thus hold the reader-writer lock in the shared mode [2]. As the checkpoint requires a consistent and quiescent FS image, it holds the lock in exclusive mode to prevent other writes from modifying the FS.

For `write`, we can see that more than half (50.98%) of CPU cycles are used on grabbing locks for accessing the file system. Exclusive-mode lock only allows exclusive access and thus yields expensive overhead due to serialization. Yet, global operations (i.e., checkpoint) are invoked by OS-wide `sync` syscall or periodically (e.g., 30s), which is usually less frequent, and hence do not significantly influence the throughput. The shared-mode lock, on the other hand, permits concurrent accesses, but its counter is shared among cores. As concurrent writes (i.e., shared-mode lock) are prevalent, the cache coherence on the lock counter value can be increasingly severe with the scaling, resulting in a considerable slowdown.

Serialization at IMDS level. After entering the file system through CC level, the process needs to access and update IMDS (i.e., the in-memory indexing and data cache). Typically, LFS tends to split IMDS into different regions (e.g., inode table, inode and data) based on functionality. F2FS manages each region via a radix tree, and uses a reader-writer lock on each tree for correct concurrent execution. As shown in Table 1, for file modification operations, such as `create` and `unlink`, they require a exclusive-mode lock as they may alter the indexing. Unlike CC level, writers can be quite popular at IMDS level. With an increasing amount of concurrent writers, serialization in accessing the three radix trees becomes severe and further leads to performance drop. From Table 2, we can see that lock operations at IMDS level are the most expensive ones with 9.89% and 37.34% in total respectively, for both `create` and `unlink`.

Serialization at SA level. Finally, to persist the data blocks in a crash safe manner, the process needs to allocate space and submit the I/O requests in the correct order. LFS typically

[1] In early Linux versions, the inode lock is implemented using a mutex.

[2] In this paper, to avoid confusing the FS reader/writer with the lock reader/writer, we refer to the shared-mode lock as the reader lock and the exclusive-mode lock as the writer lock.

uses only one logical space allocator to avoid overlapping allocation (i.e., concurrent writes on the same address). The space allocator uses mutually-exclusive locks for granting access. In this case, concurrent writes converge on the allocator, and wait to be scheduled in a serialized fashion before being sent to the device, which limits the overall throughput.

F2FS extends the single log schema into multi-head logging for data temperature separation. Specifically, for inode and data region, F2FS statically defines up to 3 types of temperature and employs multiple logs (6 log heads in total) on disk, each mapped to each temperature. However, from Table 2, lock contention at SA level can consume nearly half (i.e., 46.83%) of CPU cycles.

This is because the intrinsic dependencies among the temperature logs serialize the data persistence. In particular, for the crash consistency of F2FS, the data blocks must be durable before inode blocks and the file inode must be durable before the directory data blocks. As a result, in face of a `fsync`, these logs are almost processed serially although F2FS has multiple logs. Hence, the multi-head logging design of F2FS has little effect on scaling the I/O throughput.

3 MAX Design Principle

To exploit the benefits of multicore architecture and modern NVMe SSD, we formulate the following MAX design principles and describe the intuition behind them.

Principle 1: *In the CC level, using OS scheduler-assisted consensus to efficiently coordinate the external I/O requests (e.g., user writes) and internal operations (e.g., LFS checkpoint).*

LFS coordinates writes and checkpoint using traditional reader-writer lock. Recall that our study in §2.2 shows that the major overhead at CC level comes from cache coherence on the shared lock counter. Thus, a straightforward solution can be setting a local reader lock counter for each core, like scalable locks with per-core reader counters [41,42]. A writer, to guarantee exclusive access, can simply block all further readers, and then either aggressively query the per-core counters or await until all on-going readers finish (i.e., all counters reduced to zero). While this naïve solution successfully minimizes the cache coherence among readers, the writer may either cause excessive overhead by aggressive querying or high latency due to the lazy waiting. For instance, the inter-processor interrupt-based aggressive query [42] is likely to interfere the latency-critical tasks on other cores, e.g., increasing the user-visible latency of `read` syscall. The lazy waiting approach [41] may experience periodical OS scheduling interval (e.g., 1-10 ms) and further increase the checkpoint latency. The millisecond-scale latency may be tolerable for HDD and SATA SSD, but is unacceptable for NVMe SSDs with ten to hundreds of microsecond latency.

On the other hand, an outstanding advantage of the kernel FSes is that they run on the OS control plane. This has not been exploited to assist concurrency control of LFS. For example, to guarantee process fairness, the process in the kernel mode frequently invokes the OS scheduler for scheduling. A typical case is the exit of FS syscall, implying the end of I/O.

MAX uses a new reader-writer semaphore namely Reader Pass-through Semaphore (RPS) for concurrency control. The RPS uses per-core counters to reduce cache coherence overhead, and introduces *scheduler-assisted consensus* to coordinate the external and internal operations with less overhead.

Principle 2: *In the IMDS level, the IMDSes are partitioned by the file inode ID for concurrent file-level in-memory indexing and caching. Further, the IMDSes of the same file are repacked to avoid page-level (e.g., 4 KB) false sharing, so as to facilitate the concurrent file-level persistence functions.*

Earlier in § 2.2 we showed that serialization of accessing indexing trees at IMDS level is expensive. A feasible way to accommodate concurrent writers is to split the trees into multiple ones, similar to the non-volatile main memory FS [57]. Partitioning the memory-based storage system is relatively flexible due to fine-grained access granularity (e.g., byte-scale) of the main memory. However, such a partition method can not be directly applied to block-based SSD FS due to different and coarse access granularity. Traditionally, the FS for block storage often stitches small-sized file metadata and file system metadata into a shared block, which introduces extra contention. Therefore, scaling the IMDS while delivering flash- and concurrency-friendly on-disk layout remains a challenge for MAX.

In MAX, we propose a new IMDS abstraction, *file cell*, to repack data and metadata to allow multiple indexing entities, thereby lowering the chance of serialization. Additionally, we realize SSD- and concurrency-friendly on-disk format for file cell by setting a dedicated page for each inode and stitching the small unaligned flushes as pages. In this way, MAX can access the IMDS and write the buffered pages to the persistent storage with less contention.

Principle 3: *In the SA level, the drive space is divided into multiple independent logs; independent file operations can allocate space and be distributed to minor logs concurrently. The ordering and consistency of dependent file operations among multiple logs are delegated to crash recovery.*

Our study in §2.2 shows that having only one space allocator for each type of data can introduce considerable overhead for concurrent writers. MAX addresses this issue by partitioning the log space, like other log-based storage systems [10,57]. Log partitioning introduces challenges to maintain the concurrency and crash consistency over multiple minor logs (mlog).

For concurrency control, leveraging the file interfaces and semantics of FS, MAX distributes complete file operations (not simply data blocks) to a mlog. In other words, when persisting files (e.g., `fsync`, or `write` a file with `O_SYNC` flag), MAX submits the data blocks that need to be persisted atomically to the same mlog. This avoids concurrency control over multiple mlogs and brings higher concurrency.

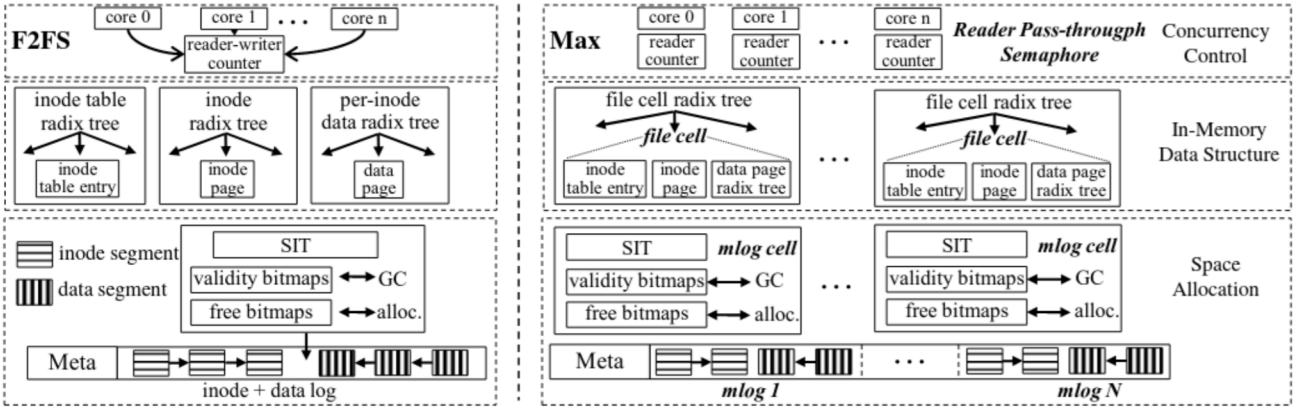

Figure 2: **A comparison of F2FS and Max.** Max *introduces Reader Pass-through Semaphore (RPS) (§4.1) at CC level, file cell (§4.2) at IMDS level and mlog cell (§4.3) at SA level for higher concurrency.*

For crash consistency, each mlog keeps its localized consistency just like traditional LFS. In Max, the same file of different versions can be distributed to different mlogs. Max embeds a global version number in each inode block to record the ordering across multiple independent mlogs, and finds the newest file during crash recovery using the global version.

By persisting independent file operations to independent mlogs and delaying the persistence ordering to recovery, Max enables highly concurrent persistent functions.

4 Max Implementation

We implemented Max by modifying F2FS. Figure 2 shows a side-by-side comparison between Max and F2FS.

4.1 Reader Pass-through Semaphore

Max replaces the traditional reader-writer semaphore of F2FS (i.e., `cp_rwsem`) with Reader Pass-through Semaphore (RPS). We use Figure 3 (referred by arrow numbers) and Algorithm 1 (referred by line numbers) to elaborate the RPS control flow.

Concurrent readers. RPS borrows the idea of per-core reader lock counter. With a private counter for each core, concurrent readers can independently increase or decrease the counter value without cache coherence from different cores (lines 1-8). The major overhead (i.e., 50.98% in Table 2) at the CC level is thus removed.

Exclusive writer. RPS introduces a "Scheduler Free Rides" mechanism to avoid high overhead and latency for exclusive-mode lock. The key idea is to leverage the CPU scheduler to efficiently check the counter value of each core. The original design goal of the CPU scheduler is to coordinate processes, which lets it frequently access the cores. Scheduling itself searches several queues, so adding RPS logic (i.e., check the reader counter) atop it costs extra little. Hence, the counter values of different cores are frequently retrieved with low overhead, thereby taking the free rides of the scheduler.

Specifically, the pending writer first locks `wsem` (i.e., Linux writer semaphore) on each core to block all further incoming readers and writers (line 15, arrows ① and ④). The writer sets the per-core notification flags, and then goes to sleep and waits for all flags cleared (i.e., readers on all cores have left the critical section, lines 16-19). Next, the on-going readers continue as usual except the last reader on each core finishes by yielding the execution to the CPU scheduler (lines 9-10, arrow ②). The CPU scheduler then clears the per-core notification flag, which indicates that the on-going readers of that core are all finished. For cores with no on-going readers, the RPS utilizes the opportunity of kernel preemption to let the scheduler check the counter value and clear the notification flag (lines 27-28, arrow ⑤). With all notification flags cleared, the scheduler can then wake up the writer and let it start to execute (lines 18-19 and 29, arrow ⑧).

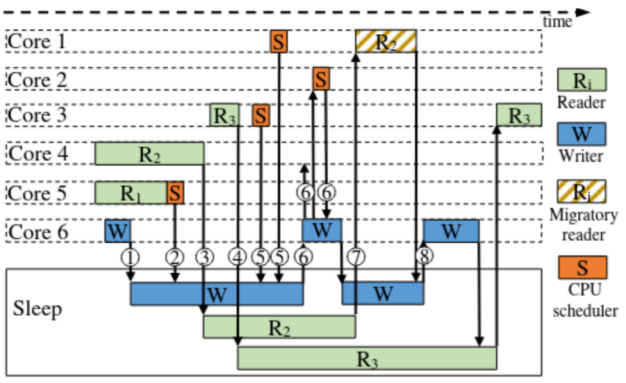

Figure 3: **"Scheduler Free Rides" mechanism of RPS.** *Described in §4.1.*

Corner cases. In our original RPS design, there are two corner cases: late preemption and task migration. First, in the "Scheduler Free Rides" mechanism, the scheduler relies on opportunities of kernel preemption to check the reader lock counter on zero-reader cores. While preemptions are usually frequent, a worst-case scenario can take up to 10 ms (i.e., tick

Algorithm 1: RPS Pseudo-code

```
1  def read_lock (rps):
2      while rps.wsem ≠ FREE do
3          wait(rps.wsem == FREE) /* woken up by line 25 */;
4      this_cpu_inc(rps.percore_reader);
5  def read_unlock (rps):
6      readers ← this_cpu_read(rps.percore_reader) ;
7      if readers > 0 then
8          readers ← this_cpu_dec_return(rps.percore_reader);
9          if rps.wsem ≠ FREE and readers == 0 then
10             yield();
11     else if readers == 0 then /* migration reader exists */
12         if atomic_dec_test(rps.migration_cnt) then
13             wake_up_writer();
14 def write_lock (rps):
15     lock(rps.wsem);
16     for core ∈ get_online_cpu() do
17         per_cpu_set(rps.noti_flag, core, 1);
18     for core ∈ get_online_cpu() do  /* wait for all flags cleared */
19         wait_timeout(per_cpu(rps.noti_flag, core) == 0, TIMEOUT);
20         if timeout then
21             cores_unfinished ← check_notification_flags();
22             send_ipi(cores_unfinished);
23     wait(atomic_read(rps.migration_cnt) == 0) /* woken up by line 13*/ ;
24 def write_unlock (rps):
25     unlock(rps.wsem);
26 def schedule (core, task):
27     if this_cpu(rps.percore_reader) == 0 then
28         this_cpu_set(rps.noti_flag, core, 0);
29         wake_up_writer();
30 def migrate_task (task, src, dst):
31     /* Tasks holding RPS are off the src and dst CPU now, so it's safe to
         modify the per-core reader counter */;
32     per_cpu_dec(task.rps.percore_reader, src);
33     atomic_inc(task.rps.migration_cnt);
```

preemption interval) for the scheduler to check all counters. This delay is unbearable for high performance SSDs and can cause a long writer latency. To handle this, RPS sets the writer to wake up periodically. Then the writer actively invokes inter-processor interrupts to check reader counters of unfinished cores (lines 20-22, arrow ⑥). Note that the wake-up interval is configurable in RPS (100 us by default).

The second corner case is the task migration. In the multicore execution environment, a reader on one core can be migrated to another core (arrows ③, ⑦). Therefore, RPS sets up a global migration counter. Upon task migration, the RPS decreases the local counter of source core by one and also increases the global migration counter by one (lines 31-33). The migrated reader therefore decreases the global migration counter instead of the local one when finishes (lines 11-12). Thus, the pending writer needs to wait until both local counters and the migration counter all decreased to zero before accessing (line 23, arrow ⑧).

4.2 File Cell

As shown in Figure 2, Max organizes the IMDSes using the file cells and indexes them by multiple trees. This subsection describes the indexing and the format of file cell in details.

File Cell indexing. Each file cell encompasses the inode table entry, inode page [3], and data page of a single file. Max then divides the file cells into multiple groups and indexes each group using a radix tree. Max places each file cell to a tree by hashing the inode ID of the file (i.e., inode ID modulo the number of trees). Each radix tree accepts the inode ID as key and outputs associated file cell. Note that the number of indexing entities (i.e., radix trees) is configurable. Having more trees yields a lower chance of serialization but can also lead to high memory consumption. We set the number as half of the number of cores as we observe that setting more trees beyond that does not lead to better performance (see § 5.4 for details). Thus, Max lowers the chances of serialization as well as the number of indexes.

File Cell data format. Max uses a dedicated page for the inode of each file. To reduce memory consumption, we use the following approach. If the unaligned data can fit in the inode page, Max appends that to the end of the inode page. If not, the inode owns the entire page. For example, consider a file with 6 KB data and a 256 B inode. Max sets two pages for that file cell. The first one is a 4 KB data page. The second one is the inode page that contains the 256 B inode plus the 2 KB unaligned data.

Unlike the journaling FS that inodes are placed in a pre-determined location, inodes of LFS are updated in an out-of-place manner, thereby forcing the inode table to be placed in a fixed on-disk location for indexing. Additionally, information similar to the inode table entries must be persisted simultaneously to locate the newest inode. However, the inode table entry is extremely small (e.g., 9 B). This brings challenges for LFSes to achieve high concurrency, low write amplification without breaking the fixed-location property.

Max realizes high concurrency and persistence of inode table entries using two representations. For in-memory representation, with the byte-addressability of DRAM, Max partitions the inode table, and distributes the inode table entries to each file cell. All operations, except checkpoint, access or modify the inode table entries inside the file cell only in the memory. Second, we reuse the classic representation (i.e., compacting different entries in pages) on disk to guarantee entries are always stored in the pre-determined locations. We rely on the periodical checkpoint to persist entries. Note that, in this design, updates on inode table entries since the most recent checkpoint can be lost in the face of a sudden crash. We further discuss how Max uses roll-forward recovery to reconstruct inode table entries in §4.4.

4.3 Mlog

Max extends the multi-head logging of F2FS by splitting the larger log into minor logs (mlogs), as shown in Figure 2. Each mlog keeps the data and inode logs each with up to three

[3] In this paper, we use the commonly-used inode table entry and inode to refer the specific node address table (NAT) and node structure of F2FS.

temperatures as in F2FS. The number of mlogs is configurable. Ideally, the number of mlogs can be the same as the number of cores to achieve the highest concurrency. However, for small capacity drives, having too many mlogs makes it hard to accommodate large files due to limited capacity per mlog. We further evaluate and discuss how to choose the appropriate number of mlogs in §5.4.

Mlog cell. To support mlog, MAX splits the allocation-related IMDSes, forming individual mlog cells. Following the design of F2FS, MAX uses segment info table (SIT), validity bitmaps and free bitmaps for space allocation. Yet, different from F2FS, MAX splits and co-locates the table with the two kinds of bitmaps in mlog cells, as shown in Figure 2.

The SIT maintains the validity of all blocks for the corresponding mlog. The free bitmap records the free 4 KB blocks. Upon data flushing (e.g., `fsync`), MAX selects a mlog cell in a round-robin fashion and searches the free bitmap for free blocks. Then, MAX sends the data to the corresponding mlog.

Garbage collection. Each mlog cell performs garbage collection (GC), i.e., victim selection and block identification and migration, independently. As GC performs at the granularity of the *section* (consecutive 2 MB *segments*), MAX keeps the validity bitmaps to record both the dirty segments that need GC, and the valid blocks per segment in the section.

For each mlog cell, MAX uses existing victim selection policies [32, 52] and the slack space recycle (SSR) [37] technique of F2FS; MAX always performs GC for inode log and we explain this in §4.4 by Figure 5. Since the data/inode blocks can be spread across different mlogs, MAX identifies not only the valid block in the mlog, but also the freshness of that block among all mlogs, by comparing the address of the block (in the mlog) with the newest address of the file data/inode; MAX migrates only the valid and newest block. If a space allocation can not find enough space in all mlog cells (e.g., the used disk volume is higher than 95%), MAX turns back to the single logging as in F2FS. While, in this case, the SA level concurrency is sacrificed, MAX avoids severe fragmentation.

4.4 Consistency

Concurrency Consistency. In traditional Linux file systems, the concurrency consistency is mostly handled by the Virtual File System. The actual file system only needs to lock on the target file(s). Therefore, with VFS remains intact, MAX simply locks on the file cells of the target files to ensure correct execution order.

Crash Consistency. After a crash (e.g., power outage), MAX recovers the state by the following two steps: (1) roll back to the latest consistent checkpoint; (2) perform roll-forward recovery on all mlogs.

Here, we use an example in Figure 4 to present the roll-forward of MAX. First, for each mlog, starting from the latest checkpoint, MAX rebuilds the inode log and forms lists of inode blocks (b.1). Next, MAX merges the per-mlog inode list.

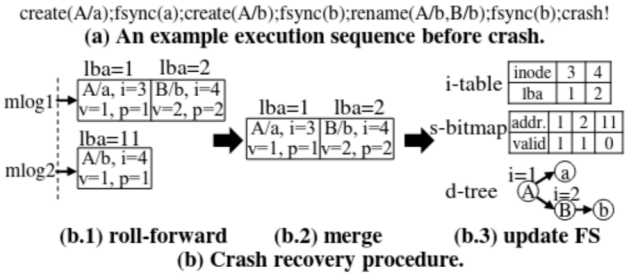

Figure 4: **An example of MAX's crash recovery.** *lba: logical block address, for recovering space bitmap (s-bitmap); i: inode number, for recovering inode table (i-table); v: global version, for merging; p: parent's inode number, for reconstructing the FS directory tree (d-tree). Described in §4.4.*

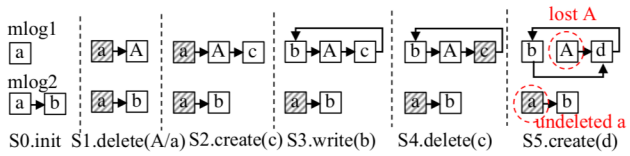

Figure 5: **Performing inode GC in SSR mode.**

Recall that MAX uses round-robin placement to writes. Therefore, outdated versions of an inode may still exist. For example in Figure 4(a), the file `b` is modified twice; `create(A/b)` is located in mlog2, and `rename(A/b,B/b)` is distributed to mlog1. Such different versions can bifurcate the merged inode list. To overcome this, MAX embeds a global version number (v) inside the inode block during each flush, and uses it to identify the latest inode block during recovery (b.2) if needed. Finally, MAX updates the inode table, the space bitmap and the directory tree with the merged inode list (b.3).

The inode list can also be affected by the GC policy. For example, using SSR, a state-of-the-art space allocation policy, can lead to an inode loop, causing consistency issues. Specifically, in Figure 5, the user deletes file `a` in directory `A`. Now, the inode of `a` is invalidated, and its parent inode `A` is updated (S1). Later when there are no more clean segments, MAX switches to SSR mode and directly reuses the obsolete blocks (e.g., `b` of S3). In this case, an inode loop is formed, and a further inode update may bifurcate original inode list, resulting in updates loss (S5). In S5, the latest `A` is lost, and the deletion of `a` is lost as well. During next roll forward, the invalid `a` in mlog2 would be considered valid. Therefore, in SSR mode, MAX still uses common GC for the inode log.

4.5 Other Important Implementations

This subsection describes how MAX's approach retains other aspects of the design of F2FS while improving concurrency.

Extent cache. F2FS uses bitmaps to record the addresses of the data blocks for a file. A bitmap-based approach is efficient for lookups of a specific point in a file but unfriendly when scanning continuous ranges of addresses. Thus, F2FS uses

per-inode extent cache to speed up address lookup (esp. range lookup). However, the per-inode extent cache is indexed by a global radix tree (`extent_tree_root`) protected by a mutex (`extent_tree_lock`), similar to the single inode radix tree. MAX splits the extent cache to each file cell for concurrent extent cache access.

Inode table journal. As we mentioned in §4.2, the inode table entry of F2FS is extremely compact (9 B). Directly updating the inode table entry to its original location is likely to incur many small I/Os, which is not friendly to performance and lifetime of flash storage. Therefore, F2FS employs an inode table journal in the spare space of the Meta region. To quickly find the dirty inode table entries that need to be journaled, F2FS uses a global linked list guarded by a spinlock (`nat_list_lock`). This introduces significant contention on the shared list when the number of threads is large. MAX only links the inode table entries for each file cell group to alleviate the contention on the list. During checkpoint, MAX scans the per-group linked list to generate the inode table journal.

Resource counters. Similar to many FSes, F2FS uses delayed allocation techniques to postpone resource allocation until the data blocks are finally sent to the persistent storage. F2FS uses many resource counters (e.g., `total_valid_block_count`) to pre-reserve FS resources for incoming I/Os. These counters are shared globally under the protection of a spinlock (`stat_lock`), and become scalability bottlenecks in the multicore environment. Actually, the FS only requires the approximate value of these counters, i.e., only needs to determine whether the incoming request fits in the FS. Hence, MAX replaces these counters with scalable approximate counters (`percpu_counter` [33]).

These modifications made by MAX do not change the write ordering, the consistent metadata format or crash recovery logic, and thus do not impact consistency.

5 Evaluation

We first evaluate MAX against state-of-the-art Linux file systems on file operations (§5.1) and applications (§5.2). Then, we perform experiments of MAX under high volume utilization (§5.3). Next, we study the performance contributions of individual design aspects of MAX (§5.4). Finally, we examine the memory consumption by running MAX (§5.5).

Testbed. The testbed is equipped with 4 Intel Xeon Gold 6140 CPU processors; each CPU has 18 physical cores (totally 72 cores) running at 2.30 GHz. The platform has 250 GB DRAM, but only 10% DRAM (i.e., 25 GB, the Linux default configuration) is used for page cache. The experiments in this section are performed on a flash-based Intel DC P3700 SSD, whose performance is presented in Table 3.

File systems setups. We compare MAX with four Linux file systems (ext4 [44], SpanFS [29], XFS [18] and F2FS [37]) in Linux kernel version 4.19.11. Ext4 and XFS are popular journaling FSes used by many Linux distributions and storage backends. SpanFS is a recent scalable journaling FS built on Ext4. We set the number of the parallel journals of SpanFS to 72, the same as the number of physical cores. All tested FSes are mounted with default options. The numbers of file cell radix trees and mlog cells are set to 36 and 8 respectively. For upper bound comparison, we use an alternative MAX-mem by disabling the `fsync` and page cache writeback functions of MAX to avoid duplicated copy.

Type	Model	Seq. Bandwidth	Rand. IOPS
NVMe	Intel DC P3700 2TB	Read: 2800 MB/s Write: 1900 MB/s	Read: 450K Write: 175K

Table 3: **SSD Specifications.**

Workloads. FxMark [48] is used to test multicore scalability. FxMark concurrently and repeatedly executes individual file operations or application processes. All tests last for at least 30 seconds and issue over 50 GB data.

5.1 Microbenchmark

5.1.1 File Operations Evaluation

Overwrite. Figure 6a shows the results of DWOL workload of FxMark. MAX achieves nearly 56× speedup at 72 cores for overwrite operations. MAX outperforms F2FS and SpanFS (the second-best) by 35× and 2× at 72-cores respectively. The key contributing factor to MAX's overwriting performance is the RPS. Overwrite operations are performed in parallel by updating individual data pages and hence do not frequently trigger page cache flushes. Thus, the major overhead occurs at the CC level. While F2FS is bounded by the CC level reader cache coherence, MAX achieves high concurrency with per-core counters in the RPS. SpanFS and XFS use multiple in-memory journal buffers, and thus serve concurrent overwrites in parallel. Nonetheless, the journaling process adds extra overhead compared to MAX.

Append write. Figure 6b reports the results of DWAL workload of FxMark. We find that MAX delivers the best performance, and achieves almost 2× the throughput of F2FS. The major contributing techniques here are the file cell and the mlog. As append writes quickly fill the page cache and trigger flushes, the I/O becomes the major bottleneck. Specifically, append writes require new data pages and new inode pages, resulting in insertions into the indexing trees. Such operations of the traditional FS incur frequent serialization at IMDS level. MAX reorganizes the IMDS with file cells and uses multiple indexing trees. Hence, the chance of serialization becomes significantly lower. Moreover, at SA level, MAX allocates space from individual mlog cells, and flushes new blocks to individual mlogs. In F2FS, the NVMe SSD is however underutilized due to the single sequential log allocation and access.

File creation. Figure 7a presents the results of MWCL workload of FxMark. We observe that, on file creation operations at 72 cores, MAX achieves 2.8× higher performance than SpanFS (the second-best) and 18.6× higher than F2FS. File creations

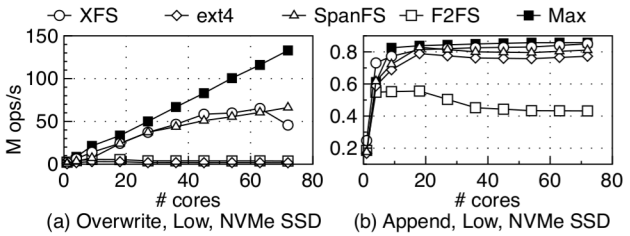

Figure 6: **Data scalability with FxMark.** *((a): overwriting blocks of private files, (b): appending blocks to private files.)*

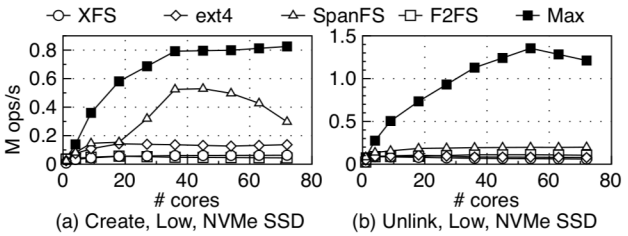

Figure 7: **Metadata scalability with FxMark.** *((a)-(b): creating or deleting empty files in private directories.)*

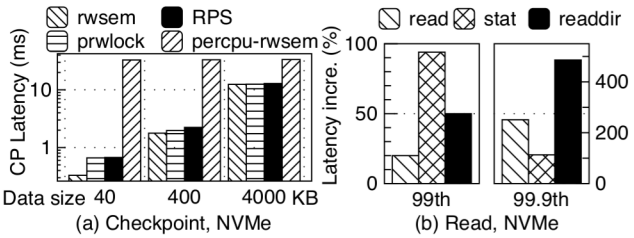

Figure 8: **Checkpoint and read latency with different lock techniques.** *(a): Checkpoint latency with different data size and locks. (b): Read long tail caused by IPIs.*

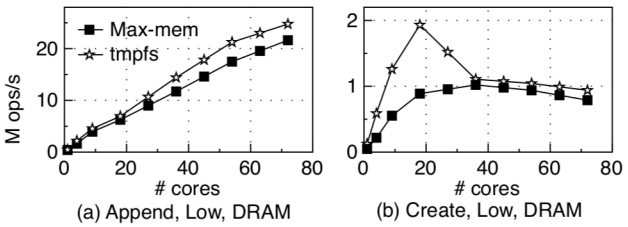

Figure 9: **Upper bound evaluation with FxMark.** *((a): appending blocks to private files, (b) creating empty files in private directories.)*

mainly consist of three steps: allocating file inode, growing directory data/inode blocks and writing back data/inode page. MAX scales well in all steps. First, file inode allocation in MAX includes the file cell allocation and insertion. MAX employs a per-core inode ID allocator and multiple file cells radix trees, thereby avoiding contention on the inode and file cell allocations; multiple radix trees also lower the chance of serialization at insertion operations to the indexing. Second, directory data and inode block grow concurrently in all tested file systems without hurting the concurrency. Third, when dirty pages are evicted from the page cache, mlog cells of MAX enable the threads to allocate space concurrently, and to distribute the dirty inode blocks to individual mlogs.

File deletion. Figure 7b shows results of MWUL workload of FxMark. MAX achieves 11.5× and 6.1× performance at 72-cores against F2FS and SpanFS (the second-best) respectively. The reasons for MAX's good scalability on file deletion are similar to that of file creation. In MAX, directory entries and inode pages are truncated independently in file cells. Also, mlog cells reclaim disk space in parallel. We observe that MAX's throughput declines since 54 cores. Further analysis suggests that the root cause is the page cache lock contention (i.e., `i_wb_list`) from VFS.

Checkpoint. We replace traditional rwsem in F2FS with several alternatives, and then collect the latencies when checkpointing variable-sized data. Figure 8a shows the results; we observe that prwlock [42] and RPS hardly affect the checkpoint latency. However, Linux percpu rwsem slows the checkpoint significantly, as its exclusive-mode lock requires RCU-based quiescence detection, where all cores have done a context switch and executed a full memory barrier.

File read. We co-run multiple foreground tasks, i.e., reading files (`read`), listing file attributes (`stat`), reading directories (`readdir`), which are conflict-free and pinned to cores, with a background task which triggers checkpoint periodically. We then re-run the above scenario without the background task and compare the results. We find that the foreground tasks are almost unaffected by percpu rwsem or RPS. However, we observe that the foreground tasks are susceptible to the inter-processor interrupts (IPIs) of prwlock. Figure 8(b) shows a performance decline: the 99.9th latency increases by up to 486%. The reason behind the long tail is that these read-dominate tasks mostly complete in a short time by hitting cache or accessing memory, which are easily affected by the forced context switch caused by IPIs.

Comparison with SpanFS. Through the aforementioned tests, we note that SpanFS scales well on single-file operations (i.e., overwrite, append write), but yields suboptimal performance on multiple-files operations (i.e., create, unlink). This is because of the global consistency maintenance across multiple journaling services. For instance, the newly created files can be distributed to a different journaling service from its parent, and the connection between the new file and its parent's directory entry (dentry) must be established which is quite expensive. MAX, due to the out-of-place update nature of LFS, can directly dispatch the newly created file along with the dentry to an arbitrary free mlog.

5.1.2 MAX-mem Performance Evaluation

We use MAX-mem to measure the performance improvement led by a sufficiently large bandwidth (i.e., using memory as backend). For comparison, we adopt tmpfs, a simple wrapper of VFS, as the theoretical upper bound [48]. Figure 9a reports

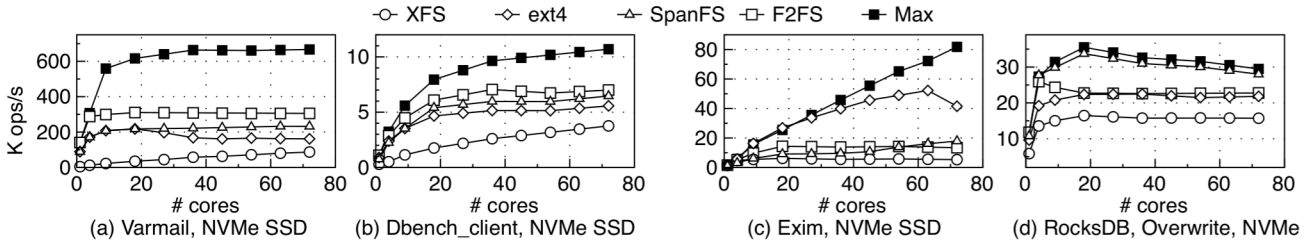

Figure 10: **Macrobenchmark.** *The workloads are write-intensive and stress underlying device. Described in §5.2.*

Workload	Average file size	# of files	Directory width	I/O size	R/W ratio
Fileserver	128 KB	10K	20	1 MB	1:2
Varmail	16 KB	1K	1M	1 MB	1:1
Dbench	client.txt, default configuration				
Exim	split spool directory, smtp_accept_max = 500				
RocksDB	overwrite, value_size=8k, disable compression				

Table 4: **Application workload characteristics.**

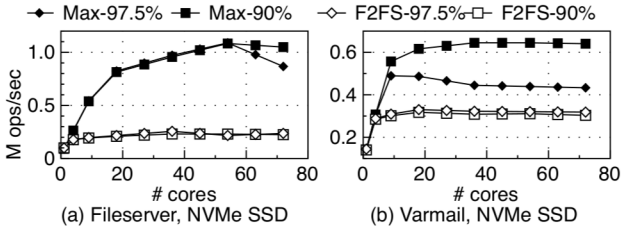

Figure 11: **Performance under high disk volume utilization.** *The number next to the file system is volume utilization.*

the results of append write operations. The throughput of both Max and tmpfs scale linearly. The little gap is caused by the overhead of supporting block storage and pre-checking the availability of FS resources (e.g., the number of data blocks).

Figure 9b indicates that the throughput of file creations of Max-mem comes close to tmpfs. Max-mem does not continue to scale mainly due to the VFS-wide spinlock `s_inode_list_lock` that serializes new inode insertions into the `s_inodes` list. We also notice the huge gap between Max-mem and tmpfs at 18-core. Tmpfs only stores in-memory states. While in Max, the dentry and inode also contain information for on-disk states, such as the dentry hash table and data block indexes. This introduces significant costs for create operation. However, when the VFS-wide spinlock kicks in, the gap becomes much smaller.

5.2 Macrobenchmark

We use Filebench [46], Dbench-client [7], RocksDB [22] and Exim [1] from FxMark to evaluate Max's scalability under applications workloads. Table 4 summarizes the characteristics of these workloads. The main rationale for choosing these four workloads is that they are both write- and I/O-intensive and can thus stress the multicore scalability.

Varmail. Varmail contains frequent metadata operations and `fsync`. Figure 10a shows the results of Varmail. Max outperforms SpanFS by 2.9× and F2FS by 1.1×. For independent file operations, Max updates the in-memory file cells concurrently with 36 indexing groups. When `fsync` is invoked, Max chooses a free mlog cell and persists the inode pages and data pages of the file cells. In contrast, F2FS uses only three shared radix trees and need to serialize concurrent threads for space allocation. Notably, SpanFS performs even worse than F2FS due to its inefficiency in handling the file creation and deletion followed by a `fsync`.

Dbench-client. Figure 10b shows the results of Dbench. Max performs best among all tested file system. F2FS stops to scale at 36 cores while Max continues to. This is because the Dbench-client performs a sequence of read, write, create, unlink, stat, rename and sync operations. For non-durable operations, such as write, create and unlink, Max delivers higher concurrency by executing operations inside each cell.

Exim. Exim focuses on small files creation and deletion. Here, we use the scalability-friendly version of Exim from the FxMark, where create and delete are performed almost in private directories. However, Figure 10c shows that only Max scale well in this modified version, outperforming F2FS by 6×. The reason for Max's good scalability is the same as file creation and deletion in §5.1.1.

RocksDB. RocksDB introduces multi-threaded flush and compaction to boost performance. We use `db_bench`, which runs four client threads to put keys in a single RocksDB instance atop a native FS. Then, we increase flush and compaction threads up to 72 threads. Figure 10d shows the result. Max outperforms its peers. Note that the throughput of Max starts to decline after 18 cores. We assume this can be caused by the following three reasons. First, RocksDB frequently invokes durability functions and hence occupies a large fraction of the device bandwidth, lowering throughput of user requests. Second, scaling over 18 cores (maximum cores of a single CPU) incurs Non-Uniform Memory Access and PCI bus routing. This throttles IOPS due to the inefficiency of remote access. Third, RocksDB manages all its files including metadata files and SSTable files under the same directory, causing a medium level contention on the shared directory.

5.3 High Volume Utilization Evaluation

We evaluate the performance under high file system volume utilization and high GC overhead in this subsection. We for-

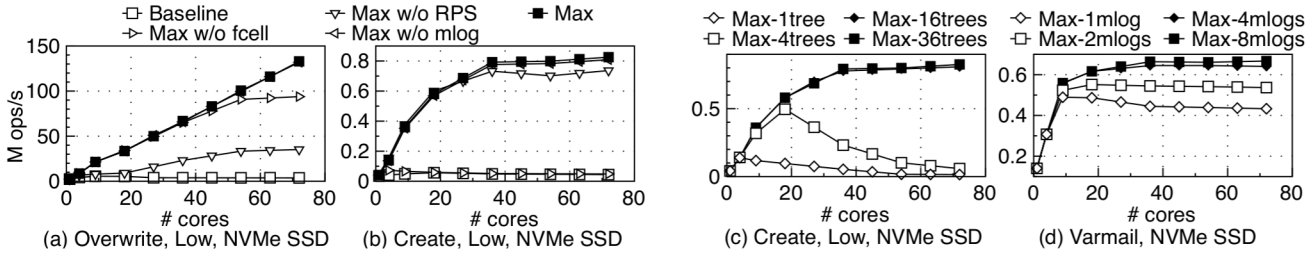

Figure 12: **Performance contributions.** *Described in Section 5.4.*

mat the 2 TB NVMe SSD and create a 200 GB partition for the test. This ensures that the device GC rarely occurs during the test. After filling up the tested file systems with two utilization ratios (90%, 97.5%), we issue another 200 GB of random overwrites to make the file system further fragmented.

Figure 11 plots the results of Fileserver and Varmail workload with FxMark. At 90%, MAX outperforms F2FS by 4.4× and 2.1×, in Fileserver and Varmail, respectively. In this case, both MAX and F2FS switch to the SSR mode. In SSR mode, besides concurrent access to file cells, MAX still offers concurrent space allocation. However, in this case, the inode updates need cleaning. The serialized checkpoint after each cleaning limits the further performance increase of MAX.

The 97.5%-utilization MAX, at 72-cores, outperforms F2FS by 3.5× and 36% in Fileserver and Varmail, respectively. We observe that the throughput of MAX starts to decline after 18 cores in Varmail. This is because MAX regresses to single logging under high volume utilization ratio. In this scenario, as RPS and file cell continue to contribute, MAX survives greater performance drop due to the single log access.

5.4 Understanding the Performance

We individually analyze the contribution of *file cell*, *mlog* and *RPS* to the performance of MAX. We setup MAX with different configurations (the left half of Figure 12): (1) F2FS (baseline), (2) MAX without file cell (Max w/o fcell), (3) MAX without mlog (Max w/o mlog), (4) MAX without RPS (Max w/o RPS), (e) full-fledged MAX (Max). The tests are the same as that in §5.1.1 and §5.2, i.e., DWOL, MWCL and Varmail.

Figure 12a-d show that, compared to the baseline, all three techniques improve the performance. Specifically, RPS has a greater boost on overwrite operations, as heavy cache coherence takes up to several milliseconds but in-memory updates only cost hundreds of nanoseconds. The file cell is more effective for complex create operations as modifying IMDS is (e.g., inode initialization and hash calculation) expensive. Mlog cell does not exhibit significant impact on Figure 12a-b because these tests are not I/O intensive.

We also study the sensitivity of MAX to the number of IMDS radix trees and the number of mlogs. Figure 12c-d plot the results of varying the number of trees and mlogs. We choose create operation for trees as it frequently modifies the indexing structure, and Varmail for mlogs due to its heavy `fsync`. We observe that MAX does not gain improvement after more than 36 trees. The major reason is that the bottleneck, after 36 trees, has shifted from file system to the VFS. For write intensive workload, 8 mlogs have almost saturated the throughput. This is because MAX, following the design of F2FS, performs checkpoint to shrink cached entries (e.g., inode table, inode). In current single-threaded checkpoint design that blocks all write operations, assigning each CPU two logs in our platform is enough currently.

5.5 Memory Consumption

We examine the extra memory consumption introduced by deploying MAX. We measure the peak memory usage for each workload when running 72 processes. Table 5 reports the result. The memory usage is categorized into two sets: (1) static: memory used by the data structures of each file system, and (2) cache: memory used by page cache. We find that MAX does not introduce much memory overhead for static memory use. At CC level of MAX, the single RPS keeps two per-core counters. At IMDS level, MAX has to maintain 36 file cell radix trees. At SA level, the segment info table and all bitmaps are physically partitioned and distributed to mlog cells, which requires no extra memory. Hence, we assume that the extra static memory usage from three levels is acceptable.

For page cache, MAX consumes more memory than F2FS at peak. This is because with file cell, the dirty pages aggregate faster to the page cache (i.e., increase memory consumption). Meanwhile, the mlog cells can also write back dirty pages faster (i.e., decrease memory consumption). As a result, extra page cache memory consumption is tolerable.

Workload	Static (MB)			Cache (MB)		
	F2FS	MAX	Incre.	F2FS	MAX	Incre.
Varmail	24.90	24.97	0.3%	1411	1484	5.2%
Dbench	24.90	24.97	0.3%	310	345	11.3%
RocksDB	24.90	24.97	0.3%	15	18	20%

Table 5: **Peak memory consumption during workload execution.** *Static: data structures, cache: page cache.*

6 Related Work

File system scalability study. Multiple studies have discussed the scalability issues in the FSes [14, 15, 17, 48]. Fx-

Mark [48] argues that file system scalability does not depend much on the storage media (i.e., Figure 1 of [48]) which is different from our observation in §2. The root cause is that, in FxMark, file operations are performed in memory with no `fsync`. Yet, when cache has little effect on absorbing I/O traffic, file system scalability *does* depend on the storage media. Such scenarios include: (1) file operations in Direct I/O mode (e.g., QEMU none cache mode [3]); (2) applications contain frequent `fsync` or `fdatasync` calls (e.g., SQLite [5]); (3) frequent cache eviction (e.g., high memory pressure).

File system scalability improvement. Another group of prior work focus on improving the file system scalability. Commuter [17] and ScaleFS [11] employ scalable software design by connecting scalability to interface commutativity. Both Commuter and ScaleFS target on a more scalable kernel (i.e., sv6) with relaxed POSIX semantics. MAX studies the multicore scalability on mature, widely-used Linux and POSIX interface. Park *et al.* [50] and Son *et al.* [56] improve the scalability of journaling. MAX focuses on LFS instead.

File system partition. SpanFS [29] and IceFS [43] partition the journaling FS. SpanFS distributes files and directories under each *domain* for scalability. IceFS partitions the file system into directory subtrees called *cubes* for failure isolation. Unfortunately, IceFS and SpanFS can incur significant overhead from maintaining the global hierarchical namespace (e.g., sharing a single physical journal in IceFS and coordination across journals in SpanFS). MAX partitions the file system at the granularity of file operations, where the file operations to mlogs are totally independent of each other.

Scalable NVMM FS. Some non-volatile main memory (NVMM) file systems [36,57] employ per-core or per-process data structures, which seem to be the direct solutions to SSD-based FS. The major difference of the scaling SSD-based and NVMM-based FS lies in the access granularity. NVMM FS can partition the FS at a finer granularity (e.g., 8 bytes); for example, NOVA [57] atomically updates the 8 B inode pointer to ensure the persistence and consistency of a file write operation. Such fine-grained partitioning is unfriendly to SSD FS with block access granularity (e.g., 4 KB). If MAX directly isolates the inode pointer (i.e., inode table entry) for each file and directly persists each pointer to SSD, MAX would suffer from huge write amplification and extra PCIe round trips. Hence, MAX introduces the file cell to repack the data structures to align the block granularity as well as to scale the file-level performance.

For concurrent space allocation, NVMM FS can partition the drive space with finer granularity, e.g., per-file log. Such fine-grained partitioning is unnecessary, or even inefficient for SSD FS. Fine-grained partition trades scalability for fragmentation. In MAX, we take a sensitivity study to find the appropriate number of logs in §5.4. Further, NVMM FS such as NOVA uses a journal to maintain consistency over multiple log partitions. MAX takes a different and more SSD-friendly approach: distributing the atomic operation to a coarse-grained log partition without spanning. To maintain the consistency over multiple log partitions, MAX embeds a global version number in each inode block instead of using a journal.

Scalable lock designs. Myriad work [16,19,21,30,31,34,39,41,42,45,47] devise scalable locks. One category [19–21,30,31,39] employ distributed reader indicators (e.g., per-NUMA node reader indicators) or similar designs to reduce the cache coherence traffic across NUMA nodes or CPU cores in the common case, which is similar to RPS. RPS uses per-core reader indicators for optimal reader-reader scalability, and uses a different "scheduler free rides" mechanism to soften the impact on the exclusive-mode lock. This mechanism aims to leverage the CPU scheduler to increase the responsiveness of the lock writers while reducing the impact (e.g., forced context switch) on other CPU cores, which we found very effective for the LFS concurrency control. RCU [47] and its extensions [34,45] allow readers just for read-only traversals. Hence, they can not be directly used in the LFS concurrency control, as readers need to perform updates. RPS maintains the number of readers in per-core reader counters, the same as in [16,41,42]. RPS differs from them in the writer procedure. Before updating protected data structures, the writer of Linux percpu rwsem [41] must wait a significantly long grace period [42]. The writer of prwlock [42] actively broadcasts IPIs to check reader status, which causes unnecessary context switches of the on-going readers. RPS leverages the CPU scheduler to retrieve reader status; the last reader of RPS voluntarily yields cores to the CPU scheduler, which enables the writer to check readers efficiently without affecting readers.

7 Conclusion

The bandwidth of SSDs has been surging over the last decade. However, through a performance study, we notice that modern Linux file systems do not offer enough multicore scalability and hence can not fully exploit the abundant bandwidth of high performance drives. We propose MAX, a multicore file system to effectively alleviate the lock contention of the file system. MAX introduces reader pass-through semaphore for efficient concurrency control, file cell for scalable in-memory data structures and mlog for concurrent space allocation. Through evaluation, we show that MAX outperforms modern Linux file systems with the scaling of cores. The source code of MAX is available at github.com/thustorage/max.

8 Acknowledgement

We sincerely thank our shepherd Ric Wheeler and the anonymous reviewers for their valuable feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), and the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300).

References

[1] Exim. https://www.exim.org/.

[2] Fdisk. https://en.wikipedia.org/wiki/Fdisk.

[3] Kvm disk cache modes. https://documentation.suse.com/sles/11-SP4/html/SLES-kvm4zseries/cha-qemu-cachemodes.html.

[4] Nvm express specification. https://nvmexpress.org/developers/nvme-specification/.

[5] Sqlite. https://www.sqlite.org/index.html.

[6] Zoned namespaces (zns) ssds. https://zonedstorage.io/introduction/zns/.

[7] Dbench. https://dbench.samba.org/, 2008.

[8] Linux perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2015.

[9] NVMe. https://nvmexpress.org/white-papers/, 2018.

[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[12] Matias Bjørling. From open-channel ssds to zoned namespaces. https://www.usenix.org/sites/default/files/conference/protected-files/nsdi19_slides_bjorling.pdf.

[13] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. 07 2013.

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[15] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[16] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[17] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[18] Jonathan Corbet. Xfs: the filesystem of the future? https://lwn.net/Articles/476263/, 2012.

[19] Dave Dice and Alex Kogan. Bravo—biased locking for reader-writer locks. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 315–328, Renton, WA, July 2019. USENIX Association.

[20] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM.

[21] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, February 2015.

[22] FaceBook. Rocksdb. https://github.com/facebook/rocksdb.

[23] Intel. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[24] Intel. Intel ssd p4618 serial. https://ark.intel.com/content/www/us/en/ark/products/series/192575/intel-ssd-dc-p4618-series.html.

[25] Intel. Intel® ssd 760p series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/

consumer-ssds/7-series/ssd-760p-series/760p-series-1-024tb-m-2-80mm-3d2.html.

[26] Intel. Intel® ssd d7-p5600 series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/d7-series/d7-p5600-series/d7-p5600-3-2tb-2-5in-3d3.html.

[27] Intel. Intel® ssd dc p3700 series. https://ark.intel.com/content/www/us/en/ark/products/series/79628/intel-ssd-dc-p3700-series.html.

[28] Intel. Intel® ssd dc p4510 series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/d7-series/dc-p4510-series/dc-p4510-15-3tb-e1-l-18mm.html.

[29] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. Spanfs: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 249–261, Berkeley, CA, USA, 2015. USENIX Association.

[30] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '19, 2019.

[31] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association.

[32] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, 1995.

[33] Linux Kernel. Percpu counter. https://elixir.bootlin.com/linux/v4.19.11/source/include/linux/percpu_counter.h.

[34] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 779–792, New York, NY, USA, 2019. ACM.

[35] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[36] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[38] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 101–110, New York, NY, USA, 2009. ACM.

[40] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with HORAE. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 549–565. USENIX Association, November 2020.

[41] Linux. Linux percpu_rwsem. http://lxr.freeelectrons.com/source/include/linux/percpurwsem.h, 2012.

[42] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 219–230, Berkeley, CA, USA, 2014. USENIX Association.

[43] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, Berkeley, CA, USA, 2014. USENIX Association.

[44] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[45] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM.

[46] Richard McDougall and Jim Mauro. Filebench. http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf, 2005.

[47] Paul E. McKenney. Kernel korner: Using rcu in the linux 2.5 kernel. *Linux J.*, 2003(114):11–, October 2003.

[48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding many-core scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 71–85, Berkeley, CA, USA, 2016. USENIX Association.

[49] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 12, USA, 2012. USENIX Association.

[50] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 787–798, Berkeley, CA, USA, 2017. USENIX Association.

[51] Christoph Rohland. tmpfs. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt, 2010.

[52] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[53] Samsung. Samsung 980pro ssd. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/980pro/.

[54] Samsung. Samsung 983 dct datacenter ssd. https://www.samsung.com/semiconductor/minisite/ssd/product/data-center/983dct/.

[55] Samsung. Ultra-low latency with samsung z-nand ssd. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[56] Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 227–240, Berkeley, CA, USA, 2018. USENIX Association.

[57] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[58] ZDNet. Zoned flash: The next big thing in enterprise ssds. https://www.zdnet.com/article/zoned-flash-is-coming/.