Manylogs: Improved CMR/SMR Disk Bandwidth and Faster Durability with Scattered Logs

Tiratat Patana-anake, Vincentius Martin†, Nora Sandler, Cheng Wu, and Haryadi S. Gunawi

University of Chicago †Surya University

***Abstract*—We introduce *manylogs*, a simple and novel concept of logging that deploys *many scattered logs* on disk such that small random writes can be appended into any log near the current disk head position (*e.g.*, the location of last large I/O). The benefit is *two-fold*: the small writes attain *fast durability* while the large I/Os still sustain *large bandwidth*.**

Manylogs also inspire a new principle: *decoupling of durability and location constraints*. For example, we can put journal blocks in *any* scattered log as they only need durability but not location constraints (*i.e.*, eventually journal blocks will be freed). We can also allow applications to specify which files that require *only* durability but *not* location constraints (*e.g.*, application commit-log files).

We show the power of manylogs for Conventional-Magnetic Recording file system (MLFS) and block-level (MLB) layers and also Shingled-Magnetic Recording (MLSMR) layer. With micro- and macro-benchmarks, compared to Linux ext3, MLFS provides up to 15x (3.7x on average) bandwidth improvement and up to 22x (5.7x) faster `sync` latency. With real-world traces, MLB and MLSMR deliver 1.9x and 1.3x I/O latency speed-up on average respectively.

I. Introduction

Drive technologies, storage usage and application I/O behaviors have evolved rapidly in the last decade. Disk capacity and bandwidth continue to increase, at a faster rate than seek improvement. Capacity and bandwidth are arguably the two main benefits that users expect to gain from disk drives; users who prioritize IOPS will seek SSD solution. Disks deployment has also evolved; it has almost entirely shifted from personal computers to datacenter servers, wherein disks are consolidated to serve multiple tenants, which is economically beneficial but introduce I/O contention.

It is a common knowledge that I/O contention disrupts disk optimal performance. A tenant with random small I/Os will significantly disrupt sequential I/Os of other tenants [18], [47], [65]. In a simple experiment we ran, 24 and 56 random 4KB IOPS can degrade disk optimal bandwidth by 25% and 53% respectively. Unfortunately, applications' behaviors in multi-tenant servers are hard to predict. A prime example comes from compute servers in public clouds. The I/O traffic to *local* disks directly attached to the compute servers (*e.g.*, ephemeral disks in Amazon EC2 instances) is typically free of charge, however, customers are warned of unpredictable performance due to fluctuating I/Os from other tenants. Using remote managed storage (*e.g.*, Amazon EBS) is an alternative, but the aggregate throughput of local disks is much higher. In short, local disks are abundant in public clouds but multi-tenancy deters users from using local disks.

To address this, we ask how local (OS-level) storage stack should evolve. Our ultimate goal is to *extract the maximum disk bandwidth in the face of random I/Os*. In achieving this goal, we argue that *random small writes are the major "enemy" of multi-tenant disks*. Large I/Os are the ideal workload as different tenants get a fair share of the disk bandwidth [47]. Random reads are problematic, but with large server caches, they can reside in memory longer (§II). On the other hand, random small writes are hard to be deprioritized in favor of large I/Os because they can originate from critical applications that require fast durability.

Modern applications unfortunately still generate random small writes. For example, tens to hundreds of `fsync` calls can be made for simple tasks [32]. Another example is NoSQL systems [62] wherein all data updates are sent to application-level commit-log files. Since these small writes do not mix well with read I/Os (introduce seeks between the commit-log and on-disk table areas), NoSQL systems tend to *sacrifice fast durability*. That is, either incoming writes are batched and users' (durable) write requests are blocked momentarily (*e.g.*, 33 ms in MongoDB) or they are buffered in OS cache and users' requests are acknowledged *before* the data is `synced` to disk (*e.g.*, 10 second `sync` period in Cassandra), which implies that a whole datacenter outage will lead to data loss [2], [8] (§IV-D). Sacrificing fast durability of small writes in favor of serving other I/Os might not be an ideal solution.

Another popular solution to handle small writes is the use of flash as a write cache, which however has some drawbacks. First, deploying an SSD in every compute server is expensive; in EC2, SSD-based hosts costs much more than disk-based hosts for the same

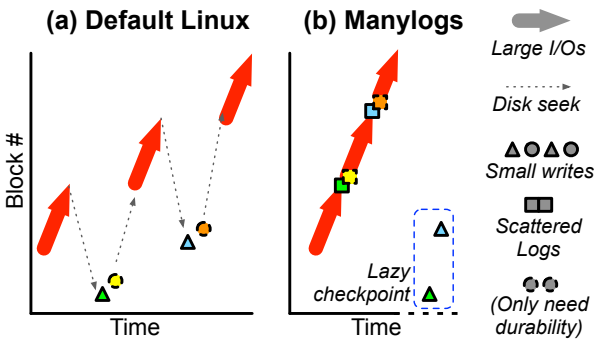

Fig. 1. **Manylogs Illustration.**

capacity. Second, SSD wears out quickly when used as a cache [37], [43], [56]. To tame this, one industry standard caches only repeated updates to recently written block numbers [4]. Here, non-overwrites will skip the flash cache and cause disk contention. From real-world traces, we observe that overwrites only range from 2-86% of all writes. NoSQL systems that always roll new commit-log files will also exhibit non-overwrites.

I-A. Manylogs

To summarize, random reads can be addressed by better caching strategies [40]. Random small writes are hard to address as they need fast durability but will hurt sequential I/O throughput. To address random writes, proposed solutions are mostly based on log-structured approach (*e.g.*, LFS [60], Gecko [65], ext3/4 data journaling) where all writes are sequentially appended to a *single log*. However, such an approach has its own historical limitations such as read-write contention and garbage collection/log cleaning overhead. To escape from expensive log cleaning, Linux by default uses "ordered journaling", a variant of log-structured approach that only logs metadata blocks, however, the non-logged data blocks will induce random writes.

In this work, we introduce *manylogs*, a simple and novel concept of logging that deploys many *scattered* logs on disk such that *small random writes can be appended into any log near the current disk head position* (*e.g.*, the location of last large I/O) such that the disk head does not seek too far or too frequent. The benefit is *two-fold*: the small writes attain *fast durability* while the large I/Os sustain *large bandwidth*.

Figure 1 contrasts the default Linux ext3/4 ordered journaling and manylogs. In Figure 1a, large I/Os (bold arrows) are interrupted by small random writes (triangles and circles) causing far and many disk seeks. In Figure 1b, our approach reserves many logs (boxes) scattered throughout the disk. In practice, there are tens of thousands of logs, ideally *one log in every disk cylinder*. In the presence of concurrent large I/Os, Figure 1b shows that random writes are made durable in the logs nearest the large I/Os (*i.e.*, "near-head logs"), hence achieving *fast durability*. At the same time, the disk head can continue serving the large I/Os without seek overhead, hence achieving *high bandwidth utilization*. Distinct to traditional logging approaches where all writes are logged, we only log small random writes, while big writes will be sent to their actual locations.

Typically, during idle periods or periodically (*e.g.*, every 5 seconds), logged writes are checkpointed to their final consistent locations (the triangles in Figure 1b). In this work, we advocate *off-hours checkpointing*, that is checkpoint can be done "very" lazily such as in off-hours (the dashed box in Figure 1b) for the benefits of foreground performance, especially in busy servers. The only drawback is that checkpoint time will be prolonged, thus reducing server availability, which can be compensated by other live replicas (§II). Checkpoint however cannot be delayed when a log is full. Thus, we introduce *log swapping* where content of a full log is moved to a cold relatively-empty log and vice versa.

Manylogs also inspires a new principle: *decoupling of durability and location constraints*. One major pitfall of storage semantic is that file system blocks have location constraints. For example, journal blocks must be written to a fixed-location log and file placement is also constrained by some heuristics (*e.g.*, related files should be nearby each other). Manylogs break this rigid rule. For example, we can put journal blocks in *any* scattered log as they only need durability but not location constraints (*i.e.*, eventually journal blocks will be freed). We can also allow applications to specify which files that require *only* durability, but *not* location constraints. A prime example is application-level commit-log files in NoSQL systems. Such files only need fast durability, but the content is *never* read except upon reboots; in normal cases, the data is flushed from in-memory tables to on-disk table files. If these application-level commit-log files are deleted or truncated, we do not need to checkpoint the blocks, as illustrated by the two circles that are not checkpointed in Figure 1b.

We show the power of manylogs at the file system (**MLFS**) and block-level (**MLB**) layers of Conventional-Magnetic Recording (CMR) drives. We also show that manylogs can be integrated to Shingled-Magnetic Recording (**MLSMR**) drives. With micro- and macro-benchmarks, compared to Linux ext3, MLFS provides up to 15x (3.7x on average) bandwidth improvement and up to 22x (5.7x) faster `sync` latency. With real-world traces, MLB and MLSMR deliver 1.9x and 1.3x I/O latency speed-up on average respectively (§V-D).

The following sections describe trends that support manylogs (§II), design and implementation of manylogs (§III) and MLFS (§IV), evaluation (§V), discussion (§VI), related work (§VII) and conclusion (§VIII).

II. Supporting Trends

We view manylogs as a simple and novel design whose time has come. In the past, manylogs would have been controversial for several reasons.

First, as many writes are redirected to logs, some mapping table is required to be maintained in a durable fashion. This is a common issue in log-structured file systems as table updates generate additional random writes [60]. Today, cheap capacitor-backed RAM (or *"cap-backed RAM"* in short) is popular [10], [35] and can be used to store mapping table safely. Upon a crash, the mapping table is small enough to be flushed to disk before the capacitor is fully discharged (§III-E). Some recent work also simplifies the problem of durable mapping table with cap-backed RAM [57], [65].

Second, manylogs seem to impose long reboot time after a crash; a full scan of all the active logs could be required to checkpoint the logged data. Such a full scan and checkpoint to empty the log is common in existing journaling file systems because log size tends to be small (*e.g.*, 128 MB). Today, with extreme disk capacity (TBs), a small fraction of disk space (*e.g.*, 5-10%) can absorb many small writes and favor lazy cleaning. Full log recovery upon reboot was also necessary when disks were a dominant storage in personal computers which typically do not employ cap-backed RAM. In contrast, with durable mapping table possible in server settings, upon reboot, we can simply read the table into memory to retrieve the redirection mapping and delay the full log recovery process.

Third, with lazy off-hour cleaning, checkpoint time is prolonged, which can reduce the disk's availability (as it is extremely busy checkpointing). However, today's Internet services are geographically distributed where each datacenter typically serves regional users, hence the strong likelihood of idle period in off-hours. In addition, as data replication is a gold standard today, server downtime can be compensated with replicas in other servers. Later we show that 1-hour delayed checkpoint only takes 1 to 117 seconds (§VI).

Finally, manylogs only address random writes but not random reads. As mentioned before, random reads can be addressed via better cache management. As an example, DULO takes into account spatial locality in addition to temporal locality (*e.g.*, LRU) [40] (*e.g.*, evicting 50 sequential blocks is more preferred than 3 random blocks). This way, repeated random reads can be served from cache. In addition, server-side memory is abundant today, with up to 1 TB of DRAM being common today [3]. We believe there is enough space for prioritizing repeated small reads. Even a 4 GB cache can hold 1 million 4-KB small reads. Therefore, we believe random writes is the major problem to solve.

III. Manylogs Design

III-A. Goals

Our goal is twofold: achieve high disk bandwidth utilization and fast durability. On one hand, large I/Os should not be interfered with seeks from small writes. On the other hand, small writes that require fast durability should not be delayed.

III-B. Scattered Manylogs

To achieve the aforementioned goals, we reserve scattered space on disk for manylogs, as already illustrated in Figure 1b. Specifically, we reserve `LogSize` of log space in every `GroupSize` of disk space (*e.g.*, 10 MB in every 100 MB). Ideally, a log space is reserved in every disk cylinder such that the disk head does not need to seek to provide durability for small writes. This design is unique compared to single-log approaches.

Manylogs can be managed at the device level (*e.g.*, disk firmware) or host level (in OS). If drive-managed, per-cylinder layout placement is possible as disk vendors know the disk internal topology. If host-managed, our implementation approximates cylinder size similar to ext3/4 group size calculation. In modern drives, small inaccuracy in predicting cylinder size (*e.g.*, a log in every 2-3 cylinders) is tolerable because modern drives can "retire multiple I/Os, spread across dozens of tracks, in a single revolution" (without seek delays) [63].

Unlike existing journaling practice, we also advocate large `LogSize`. In ext3/4 for example, the default log size is only 128 MB (only 0.01% of a 1-TB disk), which causes frequent log cleaning. Studies of disk space show that disks tend to be half full [12], [25]. Thus, it is reasonable to have a large log size. In our implementation, `LogSize` and `GroupSize` are 10 MB and 100 MB respectively (10% of disk space). Detailed justification is in Section VI.

III-C. Logged Write Size

Manylogs represent temporary durable logs for write operations whose data will be checkpointed later. For conventional (non-SMR) disks, *only* "small" writes, *not* all writes, are redirected to manylogs. For example, every write of $\leq$ 256 KB will be logged. In our evaluation, we will explore `LoggedWriteSize` threshold of 32 and 256 KB (§V-B). We also consider manylogs adoption in SMR drives (§III-I). Since SMR drives do not allow overwrites, *all* incoming writes will be redirected to manylogs.

III-D. Destination Log: Near-Head Log

To decide to which log, writes should be redirected to (*i.e.*, the destination log), we keep track the *location*

of the last non-logged I/O. In other words, this location indicates which track/cylinder the disk head is currently positioned. We call the destination log as *near-head log*. For example, in Figure 1b, the first two small writes are redirected to the near-head log (the first two boxes), near the last large I/O (the leftmost bold arrow).

III-E. Manylogs Mapping Table (MTL Table) and Capacitor-Backed RAM

All log-structured systems must maintain, for each logged block, the mapping of the log location and its final destination (*e.g.*, 100→5000 implies block #5000 is currently logged at block #100). For ease of reference, we name our mapping table "Manylogs Translation Layer" table (MTL table), akin to flash/SMR translation layer (FTL/STL) tables. MTL records block-level mapping (not extent-level) because there could be some overlapped blocks across write operations.

Decades ago, such mapping table must be stored on disk, inevitably causing performance side-effects [60]. However, we can store MTL table in cap-backed RAM (§II). Our study shows that logging ≤256-KB writes over an hour only generate 2 KB to 11 MB of MTL table across different traces (§VI), which only occupies a small portion of memory and upon a crash can be flushed to a reserved on-disk location within the range of capacitor lifetime (*e.g.*, 100-400 ms [35], [55]). The MTL Table can be flushed periodically every hour to reduce the number of dirty entries (akin to demand-based FTL page-mapping table [30]). For SMR disks, although all writes are logged, the MTL table size is interestingly not much larger than the one for conventional disks (§VI).

We do *not* use cap-backed RAM to buffer block writes due to the following constraint. Upon power failure, the RAM content must be completely flushed to disk before the capacitor discharges completely. Assuming a 100 MB/s disk and 200 ms discharge time, cap-backed RAM can safely hold only 20 MB of data (hence, we prioritize the space for the MTL table).

III-F. Lazy Checkpointing (Log Cleaning)

Eventually logged data should be checkpointed to their final locations and have their corresponding entries removed from the MTL table. To clean a log, we only checkpoint valid blocks as recorded in the MTL table. In log-based systems, checkpoint typically happens periodically (*e.g.*, every 5 minutes) or when the single-log becomes full, but this can interfere with foreground workload. Another opportunity for checkpointing is during "idle periods" [15], [50], [51], [52]. One caveat is that once the background I/Os are submitted to the disk, and suddenly foreground I/Os arrive, the OS cannot revoke the checkpoint. Thus, idle-period checkpointing can only be done gradually.

Since idle-period checkpointing has been explored extensively in literature, in this work, we explore a new idea of *off-hour checkpointing* (*e.g.*, postpone log cleaning for 8 busy hours). The intuition is that logs are rarely full when we only redirect small writes (in conventional disks); manylogs can absorb small updates over multiple busy hours without checkpointing. Off-hour cleaning can perform the entire checkpoint in a bulk. During this long checkpoint, data access can be compensated by other replicas as discussed earlier (§II).

Lazy cleaning brings two consequences. First, when a log is full, the cleaning will create a large burst of checkpoint writes. Fortunately, we can postpone the cleaning of *a* log with "log swapping", described in the next section. When *all* logs are full, massive checkpoint operations must be done, which is acceptable during off-hours (we evaluate off-hour checkpointing later in Section VI). Second, for blocks that will soon be read again after they are evicted from the cache, the latency can suffer. However, we believe such read-after-evicted-write blocks are rare with large caches.

In Linux ext3/4 file systems, blocks that are journaled but not yet checkpointed are pinned in memory. If the blocks can be evicted from memory, the file systems must ensure that future reads to the blocks are rerouted to the journal locations, not the actual ones (to read the latest version). However, these file systems do not employ a mapping table that record such information. Pinning logged, uncheckpointed blocks is a simple solution that obviates the need for a mapping table. In our case, since we already adopt a cap-backed MTL table, we can safely evict logged blocks under memory pressure. Thus, lazy checkpointing does not necessarily consume memory.

III-G. Log Swapping

To postpone cleaning of a full log, we exploit the notion of hot/cold spots. That is, there are some areas on disk where the corresponding logs are relatively empty as no I/Os are recently present in the areas (*e.g.*, inner tracks). This common condition promotes hot/cold *log swapping*. That is, the content of a full log is moved to a relatively-empty cold log and vice versa. The MTL table is also updated appropriately. The goal here is to make sure foreground performance does not fluctuate due to background random writes induced by log cleaning. Log swapping on the other hand can be done rapidly in the background as it only introduces two sets of sequential reads and writes. Moving (read+write) a 10 MB full log can be done in around 80 ms. Whenever possible, log swapping is done during idle periods when a log occupancy is above 80%.

III-H. Crash Recovery

In typical single-log journaling file systems, upon a crash, a complete checkpoint must be done for two reasons. First, the default log size tends to be small (*e.g.*, 128 MB). Thus, complete log cleaning is necessary to make room for future writes. Second, many standard Linux file systems do *not* employ a mapping table of journaled blocks. Thus, upon reboot, a full journal scan must be done wherein valid blocks have to be read from disk to the memory and checkpointed as well. In contrast, since we employ a durable MTL table (§III-E) and advocate lazy cleaning, we can simply read the MTL table from the disk in a fast sequential fashion and skip checkpoint upon reboot.

III-I. Integration

The concept of manylogs can be integrated and beneficial to different layers of storage systems: block level (of conventional disks), file systems, SMR drives, and RAID.

III-I.1 Block Level (MLB)

The most straightforward integration is to implement all of the design strategies above at the block level (for conventional disks), which we name MLB for ease of reference. This universal integration does not require changes at the file system layer. Later, we will evaluate this integration with block-level replay of real-world traces (§V-B).

III-I.2 File System (MLFS)

When manylogs are integrated at the file system layer, we can achieve more powerful capabilities such as the decoupling of durability and location constraints for journal blocks and application commit-log files, which we will elaborate more in the next section (§IV).

III-I.3 SMR Disks (MLSMR)

In commodity SMR drives, one industry standard is to use a *single* log to absorb random writes (*e.g.*, a Seagate model has a 20-25 GB persistent disk cache at the outer track [11]). To adopt manylogs to SMR drives, the proposed host-managed layout in Section III-B (*e.g.*, 10 MB log in every 100 MB) does not work because SMR drives do not allow random overwrites without rewriting the bands (*e.g.*, 15-40 MB/band [11]). However, we believe manylogs can be integrated to drive-managed SMR disks in the following novel way.

Modern disks typically have 4-6 platters, each with two active surfaces. We propose that *one* surface is manufactured as a *non-shingled platter surface* to be used for manylogs (which makes a space overhead of 1/8 to 1/12). This way, we can log random writes to any logs on the non-shingled surface where each log is essentially a circular buffer. The non-shingled surface is also required to support log swapping (without rewriting the bands). In this setup, the non-logged I/Os (§III-D) will be the read operations. Thus, the disk heads will always hover around the read areas as incoming writes can be made durable to the track on the non-shingled surface where the heads are currently aligned at.

Since the track alignments of non-shingled and shingled surfaces are different (shingled surfaces are more dense), the head on the non-shingled surface might require a small delay to find the right track (*e.g.*, in tens to hundreds of *micro*-seconds). Note that however this does not imply a seek, which can take *milli*-seconds. Overall, we believe that our manylogs-SMR integration (MLSMR) is possible and unique compared to single-log SMR drives ("SLSMR") [11]. Furthermore, our proposal of non-shingled *surface* is also different than non-shingled *regions* [13], [53].

III-I.4 RAID

Deploying manylogs in *every* drive of a RAID group will improve the aggregate RAID throughput. In our recent work [31], we show that a slow drive can degrade the performance of the entire RAID (*i.e.*, the latency of a full-stripe I/O follows the latency of the slowest drive). Such an imbalanced performance is possible in RAID where random writes affect different subsets of the drives at different times. Hence, a subpart of large full-stripe I/Os will be "the tail". Manylogs ensure that random writes to a subset of the drives do not significantly degrade the throughput of full-stripe large I/Os. In other words, manylogs attempt to make *all* RAID disks deliver similar throughput.

Integration to the RAID layer does not require changes to the RAID controller/software. This is achieved by deploying manylogs at the *aggregate* partition. For example, if we deploy 10-MB log in every 100 MB of space *on top* of 4-disk RAID-0, then on *every* disk in the RAID, there is 2.5-MB log in every 25 MB of space. We will show later that our RAID integration provides further significant speed-ups compared to single-disk manylogs integration (§V-A2).

IV. Manylogs File System (MLFS)

We now present Manylogs File System (MLFS), the integration of manylogs to the file system layer. With MLFS, we achieve two new capabilities not feasible in block-level integration. First, journal transactions can spread across scattered logs. Second, applications can specify which files require durability but not location constraints. Below, we begin with a brief overview of existing journaling modes and their tradeoffs.

IV-A. Journaling Primer

In Linux ext3/4, the two popular journaling modes are ordered and data journaling. Using similar symbols as in [22], *ordered journaling* can be represented with: D→ J_M→ M. Here, data blocks (D) are first written to their final locations, then journal transactions containing metadata blocks (J_M) must be committed to a *fixed* single-log location (*e.g.*, the beginning/middle of a disk), and later in the background the metadata blocks (M) are checkpointed. *Data journaling* can be represented with: J_{DM}→ DM. Data and metadata blocks forming a transaction (J_{DM}) are logged into the journal and later checkpointed to their final locations (DM).

The tradeoffs are the following. For small random writes, data journaling delivers fast `sync` latency as the writes are grouped into a sequential journal transaction (J_{DM}). For big writes however, the journal can be quickly full causing frequent double writes; data blocks are flushed twice, one to the journal and one to their final locations (J_{DM} + DM). On the other hand, ordered journaling is efficient for big writes (no double writes) but can suffer from long latency induced by small random writes direct to their final locations (D).

IV-B. Adaptive Journaling

To circumvent the dilemma above, *adaptive journaling* was proposed as a middle ground between the two journaling modes [58]. It works similar to ordered journaling, but for every random write in D, it is appended to the current journal transaction (akin to data journaling); a sequential write in D however is not redirected. Thus, all small random writes in D become sequential in the journal. However, they still focus on single-log journaling. In our work, we enhance single-log adaptive journaling to work on manylogs which we describe next.

IV-C. Manylogs Journaling

Adaptive journaling still suffers from the *single-log* problem which is tolerable for single-tenant file systems. In a multi-tenant server-side file system, disk head seeks back and forth between non-logged I/O areas and the single-log area. A traditional view suggests that journal transactions should be put in one circular log for the purpose of fast crash recovery; upon reboot, a sequential log scan suffices. However, journal recovery can be delayed with other alternatives (*e.g.*, read of durable MTL table suffices; §III-H) . This implies that journal blocks do *not* have to be constrained to a fixed single-log location; journal transactions can scatter across manylogs.

To support this, MLFS extends single-log adaptive journaling to *manylogs journaling* wherein every journal transaction can be written to any near-head log (§III-D). MLFS manages manylogs directly; in our MLFS-version of ext3, a log space is reserved at the end of every cylinder/block group. The log location of each transaction is stored in a capacitor-backed RAM, similar to MTL table management (§III-E).

To illustrate manylogs journaling, let's imagine a scenario where a large write L and a collection of small writes D are to be `sync`-ed. In MLFS, L is first flushed to its final location (as $L >$ `LoggedWriteSize`). Next, each small write in D ($\leq$ `LoggedWriteSize`) and other metadata blocks M form a transaction (J_{DM}) that will be put in a near-L log. In this case, the *entire* `sync` latency only incurs *one* seek (to L) and an additional rotational delay (to near-L log). In single-log adaptive journaling, the same workload will incur multiple seeks. Moreover, compared to block-level integration only (§III-I1), manylogs journaling is more superior because the small writes in D are merged in one sequential I/O in J_{DM} without a synchronization barrier; in block-level manylogs with ordered journaling on top, there is a synchronization barrier between D and J_M.

IV-D. `O_DUR`: Application Durability-Only Files

One of the core ideas of manylogs is to provide fast durability for data that does not require location constraints (*e.g.*, journal blocks as an example). However, because the OS layer does not know the semantic of application files, manylogs journaling still operates based on the `LoggedWriteSize` threshold (§III-C) for deciding which writes should be merged into a near-head log. We found an opportunity to extend this principle to the application level by allowing *applications to explicitly specify which files need fast durability but not necessarily location constraints*.

One prime example is *application commit-log files*. The last decade has seen the rise of application-level storage systems such as NoSQL key-value stores (*e.g.*, Cassandra, MongoDB, Redis). We find that all of them maintain commit-log files; all key-value updates are first appended to a commit-log file, thus preventing random writes. Note that from the perspective of the OS, commit-log files are simply regular files. Later on, the key-value updates are flushed to new table files *directly from* the application's in-memory tables, *not* from the commit-log files. For example, in Cassandra, key-value updates are reflected in "memtable" and flushed to a commit-log file, and later on the updates are copied from the memtable to new "sstable" files. This implies that commit-log files are *never read* unless there is a power failure (which triggers commit-log recovery).

This leads to our conclusion: *writes to application-level commit-log files only require durability but not location constraints*. Their content can be literally stored anywhere on disk. Only upon reboot will the content

be read; fortunately, crashes are rare [27], [61]. Furthermore, after the memtable is flushed to on-disk table files (periodically or when the table occupancy exceeds certain threshold), the current commit-log file is *no longer* needed (*e.g.*, deleted or truncated). In this case, it is *unnecessary* for MLFS to checkpoint deleted or truncated files.

More importantly, manylogs enable *fast* durability for these applications such that they do not have to lessen `sync` intensities. The problem MLFS solves pertains to the fact that *by default* commit-log file is *not* flushed to persistent storage on every write; updates are only reflected to the OS buffer cache when they return to users. By default updates are only flushed periodically. For example, MongoDB flushes every 100 ms [6], [7], Cassandra every 10 seconds [1], and Redis every 1 second [8]. In fact, some key-value stores initially did not provide flush interface to users (*e.g.*, HBase `hsync` vs. `hflush` [5]), because if one user decides to flush on every update, other tenants could suffer. Less-frequent flushes is suggested in practice because frequent log flushes cause many seeks that can hurt reads to the data files (range scan, query read, etc.). In other words, in current NoSQL practice, performance is improved by *sacrificing durability*. This is dangerous and undesirable; a whole data-center power outage leads to data loss (which happens in reality [2]).

In MLFS, our solution is to allow applications specify which files require fast durability but not location constraints. Specifically, we *extend the* `open()` *system call with a new mode,* `O_DUR`. Content of files opened in this mode will be put in manylogs regardless of the write size (*i.e.*, `LoggedWriteSize` threshold is no longer needed for `O_DUR` files). Their content is also never checkpointed (unless all logs are full). When `O_DUR` files are deleted or truncated, MLFS nullifies the corresponding entries in the MTL table to skip checkpointing of the blocks. With `O_DUR` mode in MLFS, we can maintain good performance without sacrificing durability. Later we show that we can intensify flush period to 1 ms without significantly affecting big jobs (§V-A4).

IV-E. Implementation

We implemented MLFS in 1100 LOC in Linux 3.4.77. MLFS is built on top of ext3. MLFS is appropriate to be used as a multi-tenant file system. One limitation of our current implementation is that we cannot handle multi-tenancy across multiple file systems (*e.g.*, ext3, XFS, and btrfs users on the same machine). To do so, manylogs must be implemented at the universal block layer (not only in the file system layer) and each MLFS-extended file system or guest OS should pass `O_DUR`-related information to the host's block layer. We believe these extensions are doable given the current implementation of MLFS. We leave this for future work.

V. Performance Evaluation

This section focuses on the performance evaluation of MLFS (§V-A), MLB (§V-B) and MLSMR (§V-C). We justify in detail the manylogs parameter values we use here in a subsequent section (§VI).

V-A. MLFS Evaluation

Metrics: Our two primary metrics in evaluating MLFS are (1) *sustained disk bandwidth* (percentage of the maximum disk bandwidth that large I/Os can sustain) and (2) the `sync` *latency* for small durable writes that are concurrently running with the large I/Os. We compare the performance of MLFS against three other journaling modes: Linux ext3 ordered and data journaling and adaptive journaling (§IV-B).

Hardware: We run MLFS on a Seagate Cheetah 15k.5 ST3146855FC 146GB 15000 RPM 16MB Cache Fibre Channel 3.5" disk. We also run MLFS on a RAID of four disks of the same type. The RAID is setup using RAID-0 with 64 KB chunk size. We use Linux 3.4.77 with the basic elevator I/O scheduling. The machine contains AMD FX(tm)-4130 Quad-Core Processor x86_64 2.6GHz and 8GB RAM.

Parameters: The Linux ext3 journal size is set to 5 GB (as opposed to the default 128 MB) to fairly match with the total size of MLFS manylogs. The MLFS parameters are: `LogSize` = 10 MB, `GroupSize` = 100 MB, and `LoggedWriteSize` = 32 KB. That is, we reserve 10 MB of log space in every 100 MB of disk space and only redirect writes with size $\leq$ 32 KB. In the RAID setup, the MLFS partition is striped across the *four* disks. Therefore, on *every* disk in our RAID-0, there is 2.5 MB of log in every 25 MB of space.

Benchmarks summary: We run three types of benchmarks: a set of file system microbenchmarks (§V-A1-V-A2) and macrobenchmarks (§V-A3) and HDFS+MongoDB workload (§V-A4). In all the setups, we run two types of processes in parallel: a *large sequential reader/writer* and a *random writer process*. Each of the following sections will describe the benchmarks in detail.

V-A.1 Single-disk microbenchmarks

Workload: In our first setup, the large sequential reader/writer performs 128-MB of reads/writes over a span of 2 GB file. The random writer process writes a 4-KB block to each of 20 small files scattered uniformly throughout the disk (to mimic a multi-tenant disk). After it modifies all the 20 files, it calls `sync()`. For simplicity,

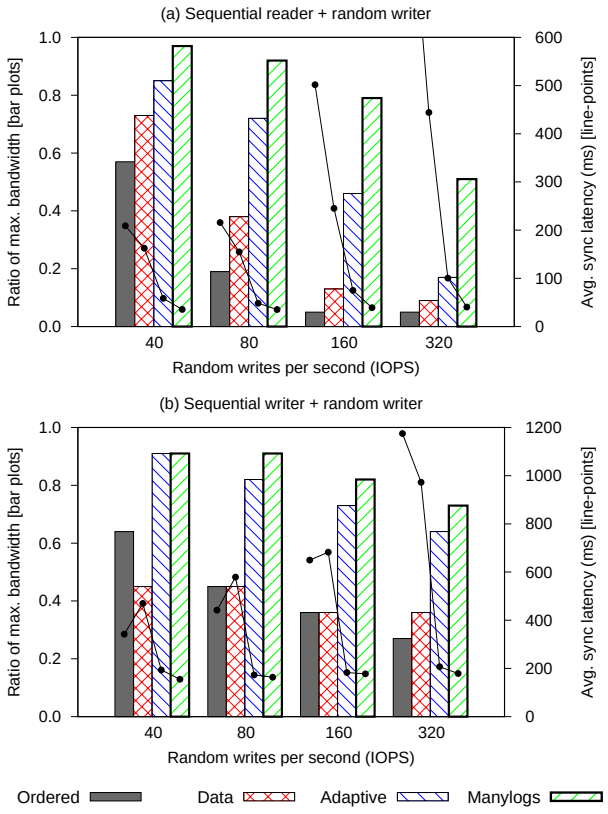

Fig. 2. **Microbenchmark results.** *We run a large sequential read/write process (top/bottom figure) concurrently with a random write process. The left y-axis (for bars) shows the ratio of the maximum disk bandwidth obtained by the sequential process. The right y-axis (for line-points) shows the average* `sync` *latency obtained by the random write process. The random writer generates X writes/per second as plotted on the x-axis. For each experiment, we compare four journaling modes: ordered (gray solid), data (red checker-box), adaptive (blue backslash-stripe) and manylogs (green slash-stripe bold-edged).*

we count 20 IOPS per `sync`. The random writer calls `sync()` at a specific rate, for example, 40 to 320 IOPS (2 to 16 `sync()` calls per second) as shown on the x-axis in Figure 2. We believe this is a realistic setup; real-world traces [9], [42] show that tens to hundreds of writes per second are common.

We emphasize one important note. To show the benefit of manylogs for large I/O throughput, the random writer's intensity *must be capped*. This rate limiting is important because as manylogs provide much faster `sync`, if we don't rate limit, random writers will flood the disk with more writes (as they achieve significant IOPS increase), preventing us to perform a fair comparison of throughput metrics. Again, our main goal is to retain large bandwidth for large sequential I/Os. Thus, our evaluation show how much bandwidth we retain given a specific maximum rate of random writers (40 to 320 IOPS in the x-axis of Figure 2).

Results: The bars in Figure 2a shows the percentage of the maximum disk bandwidth sustained by the sequential reader ("big reads") across different journaling modes and across different rates of `sync` intensity. With ordered journaling (gray/solid bars), as the random write intensity increases from 40 to 320 IOPS, the throughput of large reads collapses from 57% to 5% of the maximum disk bandwidth. This is because all data blocks are sent to their final locations, hence creating a burst of random writes. The condition is better with (single-log) adaptive journaling (85% to 17% sustained bandwidth as shown by the blue/backslash-stripe bars). Here, small random writes are merged into the journal, however adaptive journaling still suffers from the *single-log* problem (§IV-B), causing the disk head to seek back and forth between journal and read areas. On the other hand, *manylogs journaling results in the highest sustained bandwidth* (97% to 51% sustained bandwidth as shown by the green/bold-edged bars) simply because `sync`ed data can be absorbed by scattered manylogs, allowing the disk to alleviate disruptive seeks and continue serving the large I/Os quickly.

Not only manylogs journaling improves disk throughput, it also provides *faster durability*, as shown by the line-points in the foreground of Figure 2a (the unit is on the right y-axis). For example, in ordered journaling of 40 to 320 random-write IOPS, the `sync` average latency increases from 209 to 919 ms (the dots in the foreground of solid/gray bars). In adaptive journaling, write latency increases from 58 to 100 ms. However, in manylogs journaling, *fast durability of small writes can still be achieved even in the midst of large I/Os and high write IOPS* (35 to 40 ms as shown by the dots in the foreground of the green/bold-edged bars).

Figure 2b shows a similar experiment but this time the sequential reader is replaced with a large sequential writer. Here, MLFS is also superior to the other journaling modes. In this experiment we also see that data journaling performs worse than in the experiments in Figure 2a; this is because the double large writes problem (§IV-A).

V-A.2 Multi-disk (RAID) microbenchmarks

Figure 3 shows the same experiments we described in the previous section (Figure 2) but now MLFS runs on a 4-disk RAID-0. This setup will expose *tail latencies* in RAID (§III-I4). That is, as large sequential I/Os are striped across all disks, if one of the disks is slower than the rest, then the I/O throughput will be reduced to that of the slowest drive. Performance imbalance can occur in the presence of random writes that only affect different subsets of the drives at different times.

We can see this behavior by first comparing Figure 2a and Figure 3a with 40 IOPS/disk. In the single-disk

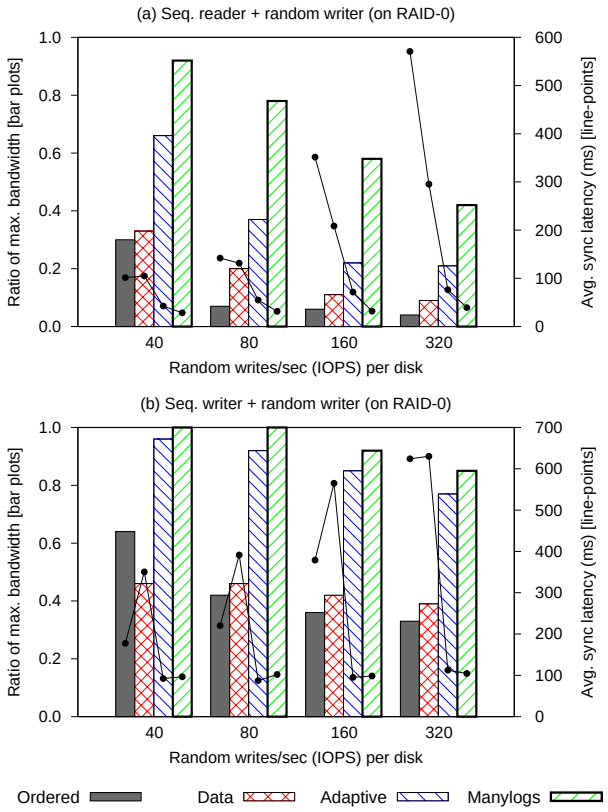

Fig. 3. **RAID microbenchmark results.** *The figures represent the same experiments as described in Figure 2, but this time the experiments run on a 4-disk RAID-0.*

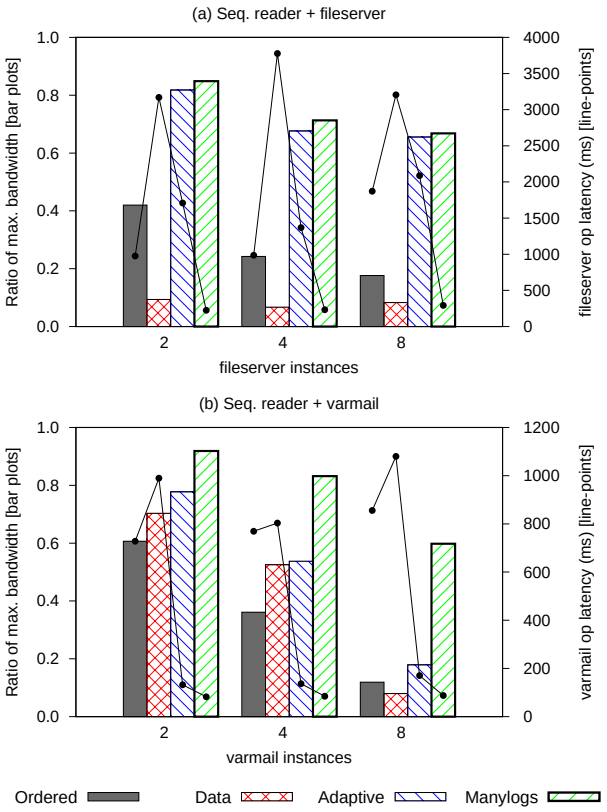

Fig. 4. **Macrobenchmark results.** *We run a large sequential reader concurrently with the fileserver/varmail (top/bottom figure) macrobenchmark, which represents the random writer process. The left and right y-axis are described in Figure 2. The x-axis shows the multiple number of instances of fileserver/varmail to emulate multi-tenancy.*

ordered journaling experiment (gray/solid bar in Figure 2a), the large I/O throughput collapses to 57%. But, in the RAID experiment (with the same 40 IOPS/disk), the throughput collapses more significantly to 30% of the maximum RAID aggregate bandwidth (gray/solid bar in Figure 3a). With the same setup, but now using manylogs, MLFS throughput does not degrade significantly (97% in single-disk and 92% in RAID experiments as shown in Figure 2a and 3a respectively). The same pattern can be observed in other IOPS/disk setups and journaling modes. Overall, our MLFS-RAID benchmarking suggests that by successfully preserving a large disk bandwidth of the individual disks, MLFS can deliver a larger aggregate RAID bandwidth.

V-A.3 Fileserver and Varmail macrobenchmarks

Workload: Now we turn to macrobenchmarks. We still run the *sequential reader/writer* to mimic Big Data workload. For the random writer, we use Filebench with *fileserver* and *varmail* personalities. The fileserver benchmark performs a sequence of creates, deletes, appends, reads, writes and attribute operations on a directory tree. 50 threads are used by default (similar to SPECsfs). The average operation latency we report for fileserver comes from the wrtfile, appendfile operations. The varmail benchmark emulates a mail server where each email is stored in a separate file. The workload consists of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory. 16 threads are used by default (similar to multi-thread Postmark). The average operation latency we report for varmail comes from the fsyncfile2, fsyncfile3 operations. We run 2, 4, and 8 fileserver and varmail instances spread uniformly on the disk. The fileserver and varmail I/Os are rate limited to 20 OPS.

Results: Figure 4a and 4b show the results for fileserver and varmail benchmarks respectively. The results show the same pattern we observed in previous experiments: in ext3, more instances (representing more tenants) lead to throughput degradation of large I/Os and longer latency of small operations. Contrary, MLFS provides the best outcomes. As the pattern is similar, for brevity, we do not elaborate the results further.

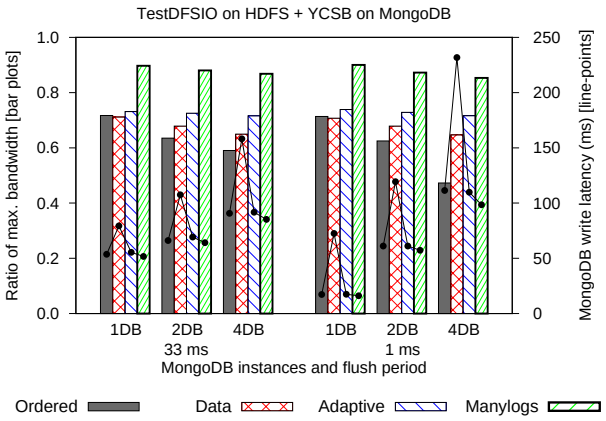

Fig. 5. **HDFS+MongoDB benchmarks.** *We run the HDFS TestDFSIO benchmark representing the large sequential I/O process concurrently with YCSB workload (key-value writes) running on MongoDB representing the random small writes. The left and right y-axis are described in Figure 2. The x-axis shows the MongoDB flush period (33 vs. 1 ms) and the number of MongoDB instances (1 to 4) we ran to emulate multi-tenancy.*

V-A.4 HDFS+MongoDB benchmarks

We now evaluate the benefits of the `O_DUR` feature (decoupling of durability and location constraints; §IV-D). We picked MongoDB as a sample application and modified MongoDB to open the commit-log file with `O_DUR` mode. This way, MLFS can redirect all commit-log writes to any near-head log. We chose MongoDB because (1) it is written in C/C++ and can easily be modified to leverage `O_DUR` by changing the `open()` system call and (2) it supports "durable write" where client requests are blocked until the data is flushed to disk; not all NoSQL systems provide durable write API.

Workload: To represent large sequential I/Os, we run HDFS TestDFSIO benchmark that reads a 3 GB file; each read call is 1 MB long. For the small random writer, we run multiple MongoDB instances (1 to 4) spread across the disk. We run YCSB [24] with 16 threads to generate key-value writes to MongoDB where each thread sends 1000 write requests per second. As mentioned above, MongoDB commit-log file is opened in `O_DUR` mode. In MongoDB, users cannot force direct `sync` on individual key-value updates. Instead, there is a background thread that will run periodically to flush the updates. Until this finishes, durable-enabled client requests will not return. This *flush period* is configurable. We use two configurations: the default value (33 ms) and a very strict value (1 ms);[1] the latter represents users who want fast durability. We use HDFS version 2.5.1 and MongoDB version 2.6.5.

[1]The actual configuration values we set were actually 100 ms and 3 ms. But, when durable writes are enabled, MongoDB divides these values by 3. We simply use 33 and 1 ms to avoid confusion.

(a)	Trace name	Description
T1	LM-TBE	Live maps back-end server
T2	DAP-DS	Caching tier for payload server
T3	DAP-PS	Payload server for ad selection
T4	Exch	Microsoft Exchange mail server
T5	CFS	MSN metadata server
T6	BEFS	MSN back-end server
T7	WBS	Windows build server
T8	DTRS	Development tools release server
T9	RAD-AS	RADIUS authentication server
T10	RAD-BE	SQL server backend

(b)	Mix description
W1	1 most-write (T7), 5 most-read (T1 x5)
W2	1 most-read (T1), 4 most-read (T7, T8, T9, T10)
W3	1 most-read (T1), 4 random (T2, T3, T5, T6)
W4	1 most-write (T7), 4 most-read (T3, T4, T5, T6)
W5	1 most-read (T1), 4 random (T2, T4, T6, T8)

TABLE I

Trace descriptions. *Table (a) describes the ten Microsoft Server traces we use [9], [42]. Table (b) shows five 5-minute workloads, each contains a mix of five traces. "most-write/read (Tx)" implies we picked a 5-minute window with the most intense write/read from trace Tx. "random" implies we pick a random 5-minute window.*

Results: Figure 5 shows six sets of experiments where we vary the number of MongoDB instances (1 to 4 DBs) and the flush period (33 to 1 ms). In each set, compared to other journaling modes, MLFS provides the best sustained throughput for HDFS and the best latency for user write requests. Most importantly, in the 1-DB setup, we can see that MLFS provides a similar throughput (89% and 90%) even when the flush period is reduced from 33 to 1 ms. This shows that *unlike common wisdom that sacrifices fast durability for the sake of better I/O throughput, manylogs break the wisdom by achieving both demands*.

We note that in the 4-DB case, request latency is worse in the 1-ms than in the 33-ms setup, which is counter-intuitive. This happens because of the too-intense `sync` forced by all the 4000 requests/second from the 4 instances. This intensity didn't happen in our previous microbenchmark and macrobenchmark experiments. The problem lies behind the serialization of the journaling layer (*i.e.*, two `sync` calls cannot be submitted to disk in parallel), which has been brought up and solved by recent work [48]. If the proposed technique is adopted, we believe MLFS can be more powerful. However, in the kernel we use, after a certain threshold of `sync` intensity, batching (*e.g.*, the 33-ms setup) gives a better overall latency.

If we extend NoSQL+MLFS integration by considering replication (*e.g.*, 3 replicas), we can leverage it by only forcing one of the replica writes with fast durability while the rest can be reflected to the application cache first and flushed periodically as in the default mode.

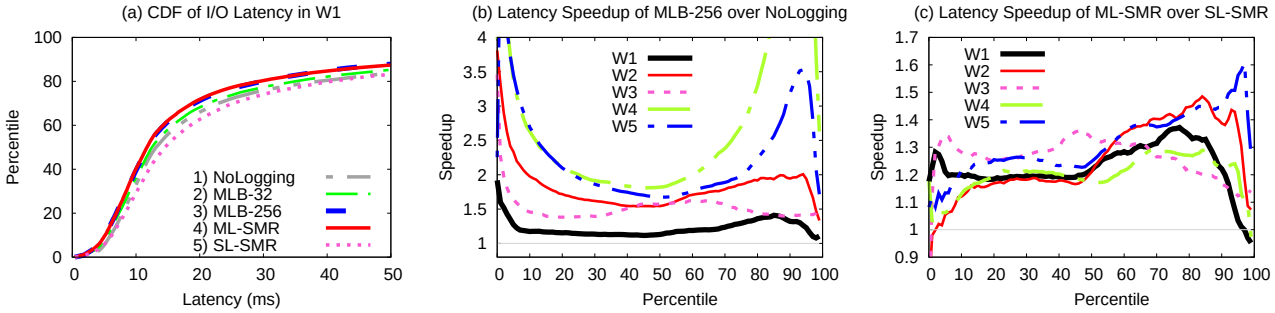

Fig. 6. **MLB and MLSMR trace driven evaluation.** *Figure (a) plots the CDF of I/O latency from replaying the I/Os in workload W1 with five strategies (NoLogging, MLB-32, MLB-256,* MLSMR*, and* SLSMR*). For brevity, we do not show the latency CDF figures for workload W2-W5. However, we summarize them in Figures (b) and (c). Figure (b) shows the speed-ups of MLB-256 over NoLogging approach at every specific percentile. For example, the black bold line (W1) represents the horizontal difference of the MLB-256 and NoLogging lines in Figure (a). Figure (c) plots the speed-ups of* MLSMR *over* SLSMR *at every specific percentile with the same style as in Figure (b).*

V-B. Block-Level Manylogs (MLB) Evaluation

We now evaluate MLB (block-level manylogs; §III-I1). MLB is less superior than MLFS as it cannot support manylogs journaling and the `O_DUR` feature. When evaluating MLB, we use *I/O latency speed-up* as the primary metric (since it is hard to differentiate "large" vs. "small" I/Os in real-world traces).

Workloads: We use 10 real-world traces from Microsoft Windows Servers [9], [42] as summarized in Table Ia. To emulate multi-tenancy (I/O behaviors from different tenants), we *mix five traces* into a single "workload", and re-rate them appropriately to prevent disk overload (as similarly done in other work [54]). Table Ib summarizes the five trace mixes (workloads W1-5) that we assembled. We make sure there is at least one mostly-read trace in the mix; write-intensive applications that demand high IOPS (*e.g.*, database transactions) should use SSD. Each workload is five minute long and replayed on the following hardware.

Hardware: We use Seagate Constellation ES.3 1TB 7200RPM 128MB Cache SATA 6.0Gb/s 3.5" Enterprise Internal Hard Drive. Disk write cache is disabled to mimic storage deployment that exercises true durability. Not disabling the cache will translate direct I/Os of the workloads above into writeback I/Os, which is not the focus of our work; we want to emulate contention from applications that require durable I/Os.

Measurement method and results: As we replay the workload, we measure the latency of every I/O. For example, the first three lines in Figure 6a show the CDFs of I/O latency in workload W1 with (1) NoLogging: unmodified trace; (2) MLB-32: manylogs with `LoggedWriteSize` $\leq$ 32 KB; and (3) MLB-256: manylogs with `LoggedWriteSize` $\leq$ 256 KB (as described in §III-C-III-D).

Figure 6a shows that MLB-32 and MLB-256 is faster than NoLogging. To quantify the improvement in detail, Figure 6b shows the latency speed-up (y-axis) at every percentile (x-axis). For example, the black bold line (W1) in Figure 6b shows the speed-up of MLB-256 over NoLogging at every percentile in Figure 6a.

Since Figure 6a only shows the results for W1, Figure 6b shows the results for W1 and W5. Overall, MLB-256 improves the I/O latency on average by 1.2-2.8x across the six workloads (the average value is taken by averaging the speed-ups at all percentiles). We will discuss later (§V-D) why this improvement is lower than the one in MLFS experiments.

We now discuss *log swapping*. Across W1 to W5, there are 17, 16, 0, 4, and 6 log swaps occurred within the 5 minute span respectively (with MLB-256). With an extreme off-hour checkpointing (*e.g.*, after 8 hours), we can extrapolate that there will only be around 1600 log swaps, far below the 10,000 logs available in a 1 TB disk (with 100 MB group size). We will measure off-hour checkpoint time in Section VI.

V-C. SMR Manylogs (MLSMR) Evaluation

Now we evaluate the performance improvement of MLSMR (manylogs SMR; §III-I3) over SLSMR (single-log SMR; *e.g.*, a Seagate model in [11]). We use the same methodology and five workloads as in the previous section. The last two lines in Figure 6a show the results of replaying the I/Os in workload W1 with MLSMR and SLSMR policies; in SLSMR all writes are first logged to the single log in outer tracks, and contrary in MLSMR, writes can be logged to any near-head log.

The horizontal distance between the MLSMR and SLSMR lines in Figure 6a show that MLSMR exhibits faster I/O latency than SLSMR. Figure 6c shows the speed-ups of MLSMR over SLSMR at every percentile. We can see average I/O speed-ups between 1.2-1.3x across the five workloads. In addition, we observe only

vs. other mode	BW Speedup	Lat. Speedup
MLFS vs. ordered	1.3-15x (3.7x)	1-22x (5.7x)
MLFS vs. adaptive	1-16x (2.7x)	0.9-7.6x (2.0x)
MLB vs. NoLogging	–	1.2-2.8x (1.9x)
MLSMR vs. SLSMR	–	1.2-1.3x (1.3x)

TABLE II

Manylogs speed-up summary. *The table summarizes the bandwidth and latency speed-ups achieved by manylogs compared to other approaches. The format of BW and latency speed-up values are: Min-Max (Average). Note that for MLB and* MLSMR*, the Min and Max values come from the minimum and maximum of average I/O latencies across the five workloads (§V-B), but at specific percentiles (e.g., Figure 6b), the speed-up can be high (e.g., 2-6x).*

64, 24, 3, 10, and 12 log swaps occurred within the 5 minute span with MLSMR.

V-D. Summary

With more than 30 experiments, we have shown how manylogs is superior than other journaling and single-log SMR approaches. Table II shows the summary of the speed-up of manylogs compared to other approaches. Overall, on average manylogs provide 2 to 5x speed-up in bandwidth and latency improvements. In some cases manylogs can provide an order of magnitude speedup.

We emphasize that our MLFS results are better than the MLB and MLSMR results, which is expected because *large* I/Os are very rare in the real-world traces we use. However, if block-level traces from modern Big Data applications are available, the benefits of MLB and MLSMR will be more apparent.

VI. Discussion of Manylog Parameters

As described in Section III, manylogs introduce some parameters to consider into storage systems design: (1) `LoggedWriteSize`, (2) `LogSize`, (3) `GroupSize`, (4) checkpoint frequency and duration, and (5) memory space for MTL table. Some of these parameters can be configured accordingly with some workload knowledge. Below we briefly discuss our experiences in configuring the parameters.

• **`LoggedWriteSize`:** Ideally, we want to redirect as many writes to manylogs provided that the logs won't be full during busy hours and hence can be cleaned in off-hours. Our evaluation (§V-B) shows that `LoggedWriteSize` of $\leq$ 256 KB is bearable. A more adaptive way is to reduce `LoggedWriteSize` on the fly when most of the logs become full before off-hours. Other partial logging approaches use slightly lower values (*e.g.*, 128 KB in adaptive journaling [58], 168 KB in proximal I/O [63], and 16 KB in flash pool [4]).

• **`LogSize`:** We reserve 10 MB of log area in every 100 MB of disk space. As disk space is typically half full [12], [25], reserving 10% of disk space for manylogs is reasonable. SMR drives have also been reported to have large logs (*e.g.*, 25 GB, 3%, of a 6TB drive [11]).

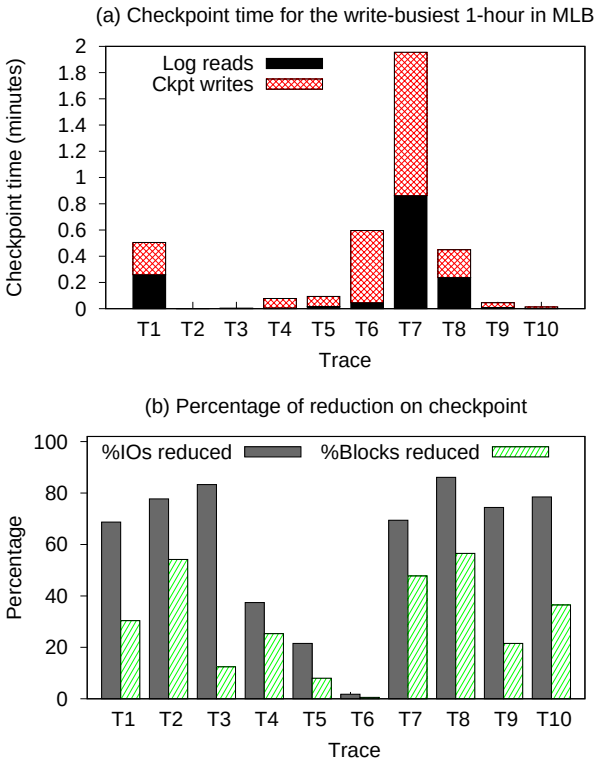

Fig. 7. **Checkpoint duration and reduction.** *Figure (a) shows the duration of lazy checkpoint after logging writes in 1-hour traces. The time is broken down to two checkpoint stages: reading the active logs ("Log reads") and checkpointing logged content to their actual locations ("Ckpt writes"). Figure (b) shows the percentage of I/Os reduced (I/Os that can be sequentially merged) and block writes reduced (blocks that are overwritten multiple times) with 1-hour lazy checkpoint, as also explained in the "Checkpoint" section of §VI.*

• **`GroupSize`:** Ideally, a log is placed in every disk cylinder. We tried 1 MB in every 10 MB, 10 in 100 MB, 100 MB in 1 GB, and 1 in 10 GB. We ran many workloads and concluded that 10 MB of log space in every 100 MB of disk space gives the best performance given our hardware. This "approximation" is roughly accurate with the fact that a common disk platter has hundreds of thousands of tracks; a simple calculation suggests that a cylinder is in tens of MB. Plus, as modern drives can "retire multiple I/Os, spread across dozens of tracks, in a single revolution" (without seek delays) [63], then a log in every 100 MB is relatively an optimum configuration. Again, this parameter depends on the underlying disk and offline profiling is needed.

• **Checkpoint frequency and duration:** Many related work suggest leveraging idle periods for background jobs [15], [50], [51], [52]. We could use idle periods to

checkpoint, but we consider an extreme case where idle periods are rare such as in busy servers. In this case, rare idle periods are better exploited for log swapping, not checkpointing. One question that we have not answered is: how long is the duration of off-hour checkpoint?

Figure 7a provides the answer. We first picked the *write-busiest* hour from each trace, T1 to T10, and ran MLB-256. (We do not show results for W1-5 because their timespan is only 5-minute long). The MTL table (§III-E) records all the block numbers that were logged and must be checkpointed after the one hour span. We then checkpoint all the logged blocks and time the duration. Figure 7a shows that lazy checkpoint duration ranges from less than 1 second to 117 seconds for 1-hour traces. For 8-hour lazy checkpoint, we can approximate by multiplying the results by 8x. Overall, this result reflects that off-hour checkpointing is viable.

We believe the checkpoint durations are quite fast in general. Figure 7b highlights the reasons. It shows two metrics: *percentages of I/Os and block writes reduced*. For the first metric, if there were two I/Os separated in time but during checkpoint they can be sequentially merged, then one I/O is reduced. For the second metric, if there were overwrites to the same block that were logged in separate places, only the latest content needs to be checkpointed, hence one block write is reduced. Overall, Figure 7b shows that by accumulating small writes ($\leq$ 256 KB) and performing checkpoint lazily, a large number of I/Os and block writes are reduced during checkpoint.

Figure 7b also points out that *manylogs can co-exist and help flash cache*. As mentioned before (§I), to extend flash lifetime, one industry standard only caches overwrites, but not first writes (within some time window) [4]. The stripe bars in Figure 7a show that overwrites range from 2 to 86%. Since flash cache will be skipped for first writes, manylogs can help improve disk performance.

We now turn our attention to checkpoint duration in SMR drives. Unlike in Figure 7a, checkpoint time in SMR drives (as shown in Figure 8) can be an order of magnitude longer (depending on the workload). This suggests that off-hour checkpointing is not entirely suitable for SMR drives. As other related work suggests, SMR log cleaning is indeed long and can take 0.6-1.6s *per band* [11].

• **MTL table size:** Finally, as discussed in Section III-E, our durable MTL table can be stored in capacitor-backed RAM. Using the same traces in Figure 7a, in one hour run of MLB-256, the MTL table size ranges from 2 KB to 11 MB across the ten traces, which are small enough to be flushed over the capacitor lifetime (*e.g.*, 100-400 ms [35], [55]). MTL table can also be periodically

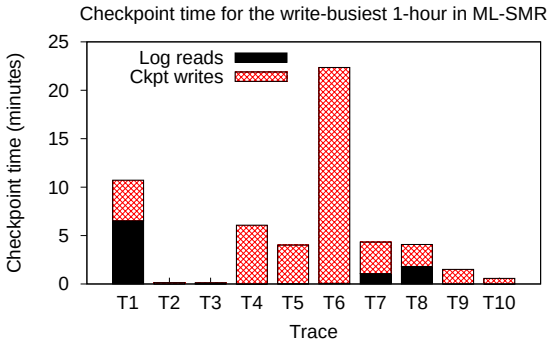

Fig. 8. **Checkpoint duration in SMR drives.** *The figure shows the same plot as in Figure 7a but this time we run* MLSMR. *Checkpoint writes are much longer as each modified band must be read and re-written.*

flushed as well (*e.g.*, every hour) to reduce dirty entries over time. We also find that with MLSMR, MTL table size is interesting similar. Further investigation shows that as all writes are logged in MLSMR, there are more overwrites (hence MTL table size does not "explode").

VII. Related Work

We now compare manylogs to other related work in the context of multi-tenant storage, log-structured storage, journaling, write optimization, block reorganization, flash cache, and SMR drives.

Multi-tenant storage: Lin *et al.* systematically show that concurrent sequential I/Os get a fair share of the bandwidth, however random I/Os will significantly hurt sequential I/O performance [47]. Unfortunately, existing studies show that random I/Os are still common [32], [46], [59]. There are many efforts to satisfy QoS in multi-tenant storage [29], [66], [67], but they typically only address one performance dimension (*e.g.*, latency or throughput). It is challenging to satisfy both random (short latency) and sequential I/Os (high bandwidth).

Log-structured: To address random reads, better cache management is the key (§II). To address small random writes, decades of storage research have shown the power of log-structured design. LFS for example treats the whole disk as a log but suffers from expensive disk seeks induced by garbage collection [60]. Recently, Gecko extended the concept to a log-structured disk array [65]. The key technique is to designate one disk in the array as a "tail" drive. All writes are first cached in SSD and then later appended to the tail disk. As all reads can be served by either the non-tail drives or SSD, read-write contention is eliminated. One major limitation of Gecko is the SSD requirement. Without SSD, all writes must be persisted directly to the tail drive, thus reducing

the array write bandwidth into a single-disk bandwidth (writes are not striped to all the disks). Contrary, our work absorbs small writes to manylogs in every disks, thus large writes still harness the aggregate bandwidth of the disk array (§V-A2).

Journaling: Journaling, the heart of many modern file systems, has received many proposed advancements. Adaptive journaling was proposed as a middle ground between data and ordered journaling [58] (§IV-B). Many recent work tackle the problem of "journaling of journal anomaly" [39], [44], [45]. For example, a custom journaling mode is allowed for every file [64] (*e.g.*, SQLite log uses ordered journaling while rollback-recovery loge uses data journaling). In another work, WALDIO reduces the excessive I/O behavior by introducing changes to SQLite and file system interface. [45]. Okeanos, a "wasteless" journaling, coalesces partial block updates from multiple writes into a single block [33]. All of these work focus on single-log journaling.

Write optimization: Besides journaling, there is an abundance of work in write optimization (*e.g.*, Bosc [49], BetrFS [38]., non-blocking writes [20], AWOL [17], no-order file systems [23]). Overall, we believe journaling and write optimizations are *orthogonal* to manylogs. That is, scattered logs can still be leveraged to provide fast durability for optimized write and journal operations.

Block re-organization: Manylogs can be viewed as a block reorganization approach. The closest related work in this space are: range writes, BORG, and WAFL. Range writes [16] allow every write (*e.g.*, supposedly to block X) to provide a block range to the disk (*e.g.*, X-m...X+n, typically a cylinder span), allowing the disk to decide the fastest placement (*e.g.*, X+2) depending on the current rotational placement of the disk head. While range writes only reduce *rotational* delays, manylogs reduce *seek* delays as small writes can be redirected to any scattered log. BORG [19] manages a small, dedicated partition on the disk as a migration destination of files in the current working set, thereby servicing a majority of I/O requests from the dedicated partition. This approach becomes challenging for Big Data applications where frequent migration can be costly. Manylogs is a more lightweight solution that only temporarily redirects writes without moving large files. WAFL [36] employs a "write anywhere" file layout. The "anywhere" placement however is a *final* placement and *constrained* on some policies best for the workload (*e.g.*, temporal locality of reference). WAFL would not work well if the "anywhere" placement is based on the disk head position (*e.g.*, in the presence of concurrent random writes, final file placements would be undesirably scattered). Manylogs however are *temporary* durable locations and do not require changes to file placement policies.

Flash cache: Host-side flash caches are becoming popular, mainly to reduce latency from the host to the back-end storage [28], [34], [43]. Flash caches are also an alternative to address small writes. For example, Proximal I/O aggregates random updates in flash until they have sufficient density to be flushed to disk in a single revolution [63]. Due to decreasing flash lifetime, caching policies must be careful. For example, to increase lifetime, an industry standard only caches block overwrites (not first writes) [4]. Flash cache suits applications that require high write IOPS (*e.g.*, database transactions). Manylogs suits workload with large I/Os and occasional small writes.

SMR drives: Finally, the advent of SMR drives calls for many changes to the file system and device layers [11], [14], [26]. For example, some work suggest reserving a small number of shingled regions [21] or a buffer of unused tracks at the end of each band [13] to persistently cache incoming writes. Some work suggest file systems to convert a portion of shingled zones into "non-shingled" (random-write) zones by introducing gaps between usable tracks [41], [53]. To support manylogs, we also suggest some portion of SMR disk to be random-writable (to support log swapping), which could be achieved by making one disk surface non-shingled (§III-I3).

VIII. Conclusion

The concept of log-structured storage has lived for decades. In this work, we introduce manylogs, a simple and novel advancement to singe-log approaches. We show that manylogs can be integrated at different layers in the storage stack and will deliver both higher disk bandwidth utilization and faster persistent writes.

Manylogs is however not a panacea. It is not designed for absorbing high intensive writes (*e.g.*, database transactions); in such cases, flash cache is more suitable. As mentioned before, checkpointing manylogs can be done in periodic, idle-based, or lazy/off-hour fashions. Our exploration of off-hour checkpointing shows that it is fitting for some but not all types of workload (§VI). Overall, it should be adopted judiciously where replicas are available and short disk unavailability is acceptable. Otherwise, background or idle-based checkpointing are more suitable.

REFERENCES

[1] Cassandra commitlog_sync interval. https://docs.datastax.com/en/cassandra/2.1/cassandra/configuration/configCassandra_yaml_r.html.

[2] Completed files lost after power failure. https://issues.apache.org/jira/browse/HDFS-5042.

[3] Dell PowerEdge Blade Servers. http://www.dell.com/us/business/p/poweredge-blade-servers.

[4] Flash Pool Design and Implementation Guide. http://www.netapp.com/us/system/pdf-reader.aspx?m=tr-4070.pdf.

[5] HBase, HDFS and durable sync. http://hadoop-hbase.blogspot.com/2012/05/hbase-hdfs-and-durable-sync.html.

[6] MongoDB Journaling. https://docs.mongodb.org/manual/core/journaling/.

[7] MongoDB Journal's Default Commit Interval. https://docs.mongodb.org/v2.6/reference/configuration-options/#storage.journal.commitIntervalMs.

[8] Redis Persistence. http://redis.io/topics/persistence.

[9] SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis. http://iotta.snia.org.

[10] Supercapacitors have the power to save you from data loss. http://www.theregister.co.uk/2014/09/24/storage_supercapacitors.

[11] Abutalib Aghayev and Peter Desnoyers. Skylight-A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[12] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[13] Ahmed Amer, JoAnne Holliday, Darrell D. E. Long, Ethan L. Miller, Jehan-Francois Paris, and Thomas Schwarz. Data Management and Layout for Shingled Magnetic Recording. *IEEE Transactions on Magnetics*, 47(10), October 2011.

[14] Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, Jehan franois Pris, and Thomas Schwarz S. J. Design Issues for a Shingled Write Disk System. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2010.

[15] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[16] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina Popovici, Aditya Akella, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Suman Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[17] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. AWOL: An Adaptive Write Optimizations Layer. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[18] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebooks Photo Storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[19] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[20] Daniel Campello, Hector Lopez, Luis Useche, Ricardo Koller, and Raju Rangaswami. Non-blocking Writes to Files. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[21] Yuval Cassuto, Marco A. A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. Indirection Systems for Shingled-recording Disk Drives. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2010.

[22] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[23] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[24] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[25] John R. Douceur and Willian J. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of the 1999 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 59–69, 1999.

[26] Tim Feldman and Garth Gibson. Shingled Magnetic Recording Areal Density Increase Requires New Data Management. *IEEE Transactions on Magnetics*, 47(10), October 2011.

[27] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[28] Binny S. Gill and Dharmendra S. Modha Center. WOW: Wise Ordering for WritesCombining Spatial and Temporal Locality in Non-Volatile Caches. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST)*, 2005.

[29] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *Proceedings of the 2011 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[30] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[31] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[32] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[33] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Okeanos: Wasteless Journaling for Fast and Reliable Multistream Storage. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, 2011.

[34] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Host-side Filesystem Journaling for Durable Shared Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[35] Gernot Heiser, Etienne Le Sueur, Adrian Danis, Aleksander Budzynowski, Tudor-Ioan Salomie, and Gustavo Alonso. RapiLog: Reducing System Complexity Through Verification. In *Proceedings of the 2013 EuroSys Conference (EuroSys)*, 2013.

[36] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter)*, 1994.

[37] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with Lazy Adaptive

Replacement. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[38] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[39] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[40] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST)*, 2005.

[41] Chao Jin, Weiya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. HiSMRfs: HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of the 30th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2014.

[42] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[43] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write Policies for Host-side Flash Caches. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[44] Duy Le, Hai Huang, and Haining Wang. Understanding Performance Implications of Nested File Systems in a Virtualized Environment. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[45] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[46] Andrew W. Leung, Shankar Pasupathy, Garth R. Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2008.

[47] Xing Lin, Yun Mao, Feifei Li, and Robert Ricci. Towards Fair Sharing of Block Storage in a Multi-tenant Cloud. In *The 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[48] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[49] Maohua Lu, Dilip Simha, and Tzi cker Chiueh. An Update-aware Storage System for Low-locality Update-intensive Workloads . In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[50] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST)*, 2002.

[51] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David Nagle, and Erik Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[52] Ningfang Mi, Alma Riska, Xin Li, Evgenia Smirni, and Erik Riedel. Restrained utilization of idleness for transparent scheduling of background tasks. In *Proceedings of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[53] Damien Le Moal, Zvonimir Bandic, and Cyril Guyot. Shingled file system host-side management of Shingled Magnetic Recording disks. In *Proceedings of the 4th International Conference on Consumer Electronics (ICCE)*, 2012.

[54] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[55] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence with Non-volatile Memories. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[56] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[57] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[58] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2005.

[59] Alma Riska, James Larkby-Lahet, and Erik Riedel. Evaluating Block-level Optimization Through the IO Path. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2007.

[60] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[61] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, 2013.

[62] Jiri Schindler. I/O Characteristics of NoSQL Databases. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, 2012.

[63] Jiri Schindler, Sandip Shete, and Keith A. Smith. Improving Throughput for Small Disk Requests with Proximal I/O. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[64] Kai Shen, Stan Park, and Meng Zhu. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST)*, 2014.

[65] Ji Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-Oblivious Disk Arrays for Cloud Storage. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[66] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[67] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.