Turn Your Storage Stack into a File System

Youngjin Kwon Henrique Fingler
Simon Peter Emmett Witchel
The University of Texas at Austin

Thomas Anderson
University of Washington

Abstract

Storage hardware trends suggest a rethink of file system design. Current and future server architectures have a multi-layer storage topology spanning multiple orders of magnitude in cost and performance. At the same time, storage devices at all layers have become firmware-managed to hide the physical shortcomings of the storage media, but inducing hidden performance and quality of service overheads. File systems should make use of this wide range of performance and capacity, while being conscious of the firmware management overheads to provide a unified interface that delivers the best of all worlds—performance, capacity, and quality of service.

We argue that such a multi-layer filesystem will be simpler to implement and to use than the complex collection of different storage systems that we have now. This is because many storage system optimizations both at the OS and application layers are designed to hide access latencies. These optimizations are less important, as devices at the top of the storage stack can provide low-latency access efficiently in hardware.

1. Introduction

File systems were originally developed to provide a convenient, common abstraction on top of individual storage devices. No matter what characteristics a storage device had, each file system on the device would support a hierarchical namespace, extensible and possibly sparse flat files, metadata crash consistency, common dynamic mount points for combining file systems across multiple devices, etc., all while hiding the device specifics from the user (other than device capacity) [22]. And because storage devices became, over time, increasingly slow relative to processors and main memory, file system designers added complex implementation strategies to hide latency, including deep operation pipelines, fine-grained locking, deferred block allocation, and complex failure recovery mechanisms [12, 15, 21]. Clumsy failure semantics for application data, while not a design goal, became an inevitable byproduct of the file system designer's need for device independent performance optimization [13].

Recent hardware developments suggest a rethink. Latency to persistent storage is rapidly falling. Per-node and disaggregated flash have become nearly ubiquitous within the cloud, providing a scalable, low latency alternative to disk. Even faster non-volatile devices are beginning to be available, with some able to perform durable writes within 100ns, five orders of magnitude faster than magnetic disk. Disk retains a cost-capacity advantage over flash, and flash over non-volatile memory, suggesting future systems will have deep storage hierarchies. Finally, many storage devices have become internally managed due to physical constraints, such as the need for flash wear-levelling or shingle writes to optimize magnetic disk density. As a result, the performance of generic file systems on these new devices often substantially lags the performance of the underlying hardware [25].

We believe it no longer makes sense to engineer file systems around the assumption that file operations are inherently slow, that the kernel intermediates every file system metadata operation, or that each file system is tied to a single physical device.

Some advocate that a cloud persistent block store can take the place of file system storage for most applications, transparently managing the various devices in the storage hierarchy for the user. We believe, however, that file systems still have a role to play, both in the cloud and beyond – they are not just a vestige of an old, forgotten time. File system namespaces provide a well-defined and easy to understand abstraction for users, and an interoperability layer for applications, not easily reproduced in persistent block stores.

Nor do we think it cost-effective or performant to layer multi-level file systems on top of per-node and per-device file systems. Once the file system is inherently multi-level, various parts of the storage design, including lookup, block allocation, and persistence, can be substantially simplified at lower levels and tuned to take advantage of device specific properties. There is a substantial benefit to be recouped by removing that indirection.

We present the design of MLFS, a multi-level file system that achieves the full performance of each layer of the storage hierarchy. The fastest layer requires kernel bypass in the common case for both file data and metadata, optimizing for low latency, but recoverable, user-level file operations. This also allows us to provide precise, synchronous write semantics for both user data and file system metadata. Once persistent, writes are digested by the kernel in the background and made available to other

Memory	Latency	Throughput	Cost/Byte
DRAM	100 ns	80 GB/s	1000x
NVM	1 μs	10 GB/s	100x
SSD	10 μs	10 GB/s	10x
HDD	10 ms	100 MB/s	1x

Table 1. Future server memory hierarchy (figs. approx.) [30].

applications. Digests are device-aware, using block sizes that remove device-layer garbage collection overhead, for example, by writing full shingles on a shingle disk and full erasure blocks on a flash device. Finally, because data blocks vary in size across layers, digests reorganize and compact, making lookups efficient.

This paper makes the case for such a file system design. We start by providing more background on current storage technologies and their use in the cloud (§2). We then provide a design sketch of our file system (§3) and discuss several use-cases (§4).

2. Background

Fast persistence. With the introduction of high-density flash and other non-volatile memory technologies, recent years have seen a staggering race towards lower latency and higher throughput. To support this performance, many devices now make use of hardware memory and I/O virtualization technology (SR-IOV [17] for SSDs and the MMU for NVM) that allow for kernel-bypass.

This trend has had an impact on how file systems achieve persistence because the old assumption of slow storage devices no longer holds. Existing persistence optimizations have become obsolete, while the cost of indirection introduced by some of these optimizations, such as kernel-mediated buffer caches, now limit rather than help performance.

Today's server storage hierarchy. At the same time, storage technology has evolved from a single viable technology (that of the hard disk drive) into a diversified set of offerings that each fill a niche in the design tradeoff of cost, performance, and capacity. Three storage technologies stand out as stable contenders in the near-future for data center servers: Non-volatile memory (NVM), solid state drives (SSDs), and high-density hard disk drives (HDDs). While HDDs and SSDs are already a commodity in servers today, NVM is expected to be added in the near future (most likely based on 3D XPoint technology). Table 1 shows each technology and its expected long-term place in the design space. We can see that each trades off one design feature for another by at least an order of magnitude.

Hardware storage management. To get the most performance from their devices, hardware manufacturers have embraced techniques that require complex firmware management. Flash memory requires a translation layer in order to manage erase cycles and wear. Shingled magnetic recording requires management to enforce sequential writing of shingle zones.

Firmware abstracts away the limitations of the hardware and provides a generalized block interface to the file system, but it does so at a cost. For example, Facebook reports a throughput slow-down of up to 6$\times$ when data is written randomly to an SSD due to firmware management overhead [31]. To improve performance and predictability, special interfaces are provided by storage controllers for certain write patterns. For example, the NVMe interface specification provides the dataset management command allowing the user to specify I/O patterns for regions of an SSD, such as sequential access [4]. The SCSI and ATA interface specifications are extended with zoned block commands for shingled disks that specify sequential write preferred and required zones [29].

Fine-grained persistence. As storage technology evolves, applications morph to require more fine-grained persistence. Often, files are merely named address spaces that contain many internal objects. This transformation has happened both at the edge [13], as well as in the data center through the use of key-value stores, data base backends, such as SQLite [5] and LevelDB [3], revision management systems, and distributed configuration software, such as Zookeeper [1]. Due to these fine-grained persistence requirements, applications are employing a wide range of complex update protocols, making them vulnerable to bugs and errors [27].

Call to arms. Existing file systems poorly address these developments. Abstractions are required to manage the diverse hierarchy of server storage technology, but file systems today manage each layer of the hierarchy independently, with specialized interfaces, no notion of cost-per-byte trade off among the different layers, or mechanisms to migrate data among layers. Applications must explicitly move data among a variety of devices and services, resulting in complex, ad hoc techniques that need to be manually updated whenever underlying storage or pricing conditions change.

3. MLFS Design

By designing a file system that manages data across modern storage devices, we can combine their strengths while compensating for their weaknesses. A proper file system requires more than simply moving objects between storage layers according to replacement policies. Our file system design spans storage devices, provides performance superior to current file systems, and strives to provide the following benefits that have proved difficult to provide in previous file systems:

- **Fast writes.** Our file system must support fast, random writes. An important motivation for fast writes is supporting networked systems which must persist data before issuing a reply.

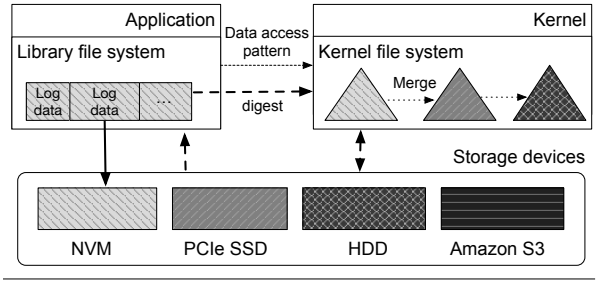

Figure 1. MLFS design. Solid arrows are synchronous I/O. Dashed arrows are asynchronous I/O. Triangles are LSM trees, stored on the various storage devices.

- **Syncless consistency.** Today's file systems create a usability and performance problem by guaranteeing persistence only in response to explicit programmer action (e.g., `sync`, `fsync`, `fdatasync`). File systems use a variety of complicated mechanisms (e.g., delayed allocation) to provide performance under the assumption of slow device persistence. Data should be persistent after a `write` and MLFS can provide that guarantee without sacrificing performance.
- **Device-specific optimizations.** Storage devices have device-specific quirks that have a first-order effect on performance, wear, and quality of service. Examples include the flash translation layer in SSDs and the shingle size in shingled disks. Managing these quirks allows us to reduce write amplification and garbage collection and thus minimize their effect on performance and wear. Further, these optimizations are simpler once we can assume that all data being migrated between layers is already persistent.
- **Unified interface.** We provide a unified file system interface to the entire underlying storage hierarchy. MLFS is backwards compatible to existing POSIX applications but also allows application developers to specify performance and cost requirements for storing data that are managed transparently by MLFS.

Figure 1 shows a high-level overview of the MLFS design. To achieve our design goals, we integrate a number of design ideas.

Log-structured merge trees. MLFS structures all data as a set of log-structured merge (LSM) trees [24]. LSM trees separate read and write paths and are a first step to provide fast writes and syncless consistency. Applications write data to an operational log resident in NVM for low-latency persistence (*logging*). Logging naturally provides low-latency durability; it forces operations to be serialized, provides data consistency, crash recovery and can even ensure atomicity of operations. Logs are highly desirable for writing, but cumbersome to search and read. Thus, MLFS periodically *digests* logs into read-optimized formats. Digests happen asynchronously and the log is garbage-collected.

Sequential, aligned writes. Another benefit of digesting writes in bulk is that the read-optimized version can also be written sequentially, minimizing the impact of device management firmware, such as garbage collection. MLFS enforces all LSM writes to be sequential and aligned to large block boundaries that are efficient for the device, such as erasure blocks for SSDs and write zones for shingled disks. When data is updated, old versions are not immediately deleted and overwritten. Instead, MLFS periodically *merges* multiple copies of read-only data to reclaim free space in multiples of the device block size.

User/Kernel division of labor. MLFS has a novel division of labor between user-level code and the kernel. In order to attain fast writes, we separate the responsibilities of logging and digesting/merging and assign them to user-level software and the kernel, respectively. User-level applications are granted direct access to a small private space on a device for efficient logging as well as direct read-only access to cached portions of the global LSM-tree space.

The kernel is responsible for digesting and merging of read-only data. It is at the digest stage that MLFS enforces metadata integrity and makes data available globally. Upon a crash, the kernel is responsible for recovering file system state. Doing so is not different from its regular operation. On boot it digests each application's log. This finds and makes globally visible each application's consistent state.

Hardware virtualization. To bypass the kernel efficiently, we make use of the hardware virtualization capabilities available in modern SSDs and NVM. This assumes that we can restrict each application's access to contiguous subsets of each device's storage space, according to access rights. The MMU trivially supports this feature for NVM, while NVMe provides it via *namespaces* that can be attached to SR-IOV virtualized SSDs [4]. Bypassing the kernel moves all performance-sensitive mechanisms of the file system into a user-level library, where they can be customized to an application if beneficial [26]. HDDs do not require kernel bypass.

Data access pattern API. To provide a unified interface that takes advantage of the entire storage hierarchy's performance and capacity, the kernel transparently migrates data among different storage layers. Because the kernel is bypassed at the top layer, MLFS exports an API that applications use to communicate access patterns to the kernel. Hot blocks are kept in higher-performing devices. The kernel gathers access patterns and can prioritize it fairly according to administrator-set application priorities and allotted storage quotas. Applications can misuse the APIs but doing so only harms their own performance.

	EXT4-DAX (SYNC)	EXT4-DAX	MLFS
Write seq.	26.5	3.7	2.3
Write rand.	26.4	4.2	3.4
Overwrite	26.5	4.0	3.3
Read seq.	0.9	0.9	0.8
Read rand.	2.4	2.4	2.3
Read hot	1.7	1.6	1.5

Table 2. Latency [μs] for LevelDB benchmarks. SYNC means synchronous writes. Read hot is a workload that reads 1% of the hot data in a database in random order.

Data access leases. While we expect fine-grained sharing to continue to be mediated by the kernel, sequential sharing can be managed by leases. Similar to their function in distributed file systems [14], leases allow an application exclusive access to the blocks specified in the lease. As long as a lease is held, an application may write to these blocks without kernel mediation, while operations from other applications are serialized before or after the lease period. Leases can expire and may be revoked by the kernel at any time. It is the application's responsibility to revert written state in this event.

4. Use cases

This section explores MLFS's potential to significantly enhance the performance and robustness of a number of important cloud applications.

Big data. Data-intensive applications, such as graph processing, read and write petabytes of data, yet demand the highest-performance access to their data sets. The simple act of writing or scanning a large file at the highest possible speed, while storing it at the lowest possible cost requires coordinated access to all levels of the storage hierarchy.

Current file system designs do not abstract the entire storage hierarchy, leading to complicated application-level mechanisms that allocate and migrate data among the different storage tiers. Less complex applications simply choose one storage tier for all data storage and live with a bad tradeoff between performance and cost.

Embedded databases and key-value stores. Embedded databases like SQLite [5] and key-value stores like LevelDB [3] use the file system for data storage, but export interfaces that allow applications to process and query structured data. Applications built on these interfaces make extensive use of the underlying file system, and therefore present an opportunity for MLFS.

Table 2 shows the latency of several LevelDB microbenchmarks when run on an early prototype of MLFS that supports logging and digesting and EXT4-DAX, a version of ext4 that directly maps non-volatile memory to user-space. EXT4-DAX is run in both synchronous and asynchronous modes (where synchronous mode provides the same persistence guarantees as MLFS). The underlying storage device is DRAM, which does not have the same timing as NVM, but all configurations use the same device. This is favorable for EXT4-DAX, because EXT4-DAX has many random writes that are not penalized by management firmware overhead.

The table shows that MLFS has the lowest latency for all operations. For writes, MLFS is up to 38% faster than EXT4-DAX with asynchronous I/O—while providing much stronger durability guarantees—and up to 11.5$\times$ faster than EXT4-DAX with synchronous I/O.

Our experiment demonstrates that a file system with a simple, synchronous I/O interface can provide high performance (if the underlying storage device is fast). It is far easier to do crash recovery when writes are synchronous and in-order. Modern applications struggle to make logically consistent updates that are crash recoverable [27]. MLFS can help such systems by providing simple recovery semantics. SQLite must call `fsync` repeatedly to persist data to its log and to persist its data file so it can reclaim the log. Many or all of these `fsync`s would become unnecessary if data is written synchronously.

Fast RPC. Many data center server applications rely on fast remote procedure calls (RPC) to work efficiently. Key-value stores, distributed consensus mechansisms, and web servers are just a few examples. In each of these cases, the applications also need to access persistent storage within the execution timeframe of the RPC. For example, distributed consensus mechanisms, such as Paxos, must locally persist consensus outcomes across several nodes before acknowledging the consensus. Low latency storage is often the determining performance factor in these applications. By providing low-latency, synchronous I/O semantics, MLFS reduces the need for complex latency hiding techniques while speeding up RPC-based systems at the same time.

Public Cloud. In the public cloud, multiple tenants operate on shared physical storage, often via NFS-like file systems. In these scenarios, quality of service (QoS) guarantees and device longevity are important assets to the cloud provider, keeping tenants isolated while reducing cost.

Current cloud storage stacks do not provide a good handle on either QoS or device wear, as both are impacted by write amplification and garbage collection. This leads cloud providers to under-utilize their storage media to minimize contention among tenants and to use complex cost-models to predict it [28]. By writing aligned, erasure and shingle-sized blocks, MLFS can minimize device write amplification. This allows the cloud to better enforce QoS under shared operation even under high utilization. It also allows the provider to transparently migrate data among similar storage options to equalize wear, leading to lower total cost of ownership.

Transactions. Because MLFS synchronously updates a write-ahead log, it can naturally support transactions for file state because operations can be recovered after a crash. Transactions are a programmer-friendly abstraction for querying and updating structured data because they provide easy-to-program, crash-consistent semantics. Version control systems like git and Mercurial store their data in the file system using a variety of techniques (e.g., lock files, logging, manifest). It is currently difficult for these systems to make crash-consistent updates to their state. Transactions would make it easy. Finally, consider an editor that must create a temporary copy of a file being edited so it can atomically rename the temporary file to its permanent location when the user saves the file. Transactions eliminate the need to copy the file.

MLFS transactions would have some natural limitations. Their size would be limited by the capacity of the fastest storage tier. Also, application logs are not shared, so MLFS supports transactions between multiple threads, but not multiple processes.

5. Related Work

Managed storage designs. All storage hardware technologies require a certain level of software management to achieve good performance. Classic examples include elevator scheduling [2] and log-structured file systems [7]. Modern examples include log-structured merge trees [24] (LSM-trees), used by the Anvil [20] modular storage system. All of these systems rely on a particular layout of the stored data to optimize read or write performance or (in the case of LSM-trees) both. We build on this work to optimize data layout for the storage technologies of today's data center servers.

File system API. A number of approaches propose to redesign the file system interface to provide additional performance to certain applications. Rethink the sync [23] proposes the concept of *external synchrony*, whereby all file system operations are internally (to the application) asynchronous. The operating system tracks when file system operations become externally visible (to the user) and synchronizes operations at this point. While this worked well in the personal computing world, where a single operating system controls all user-visible output, it is prohibitively complex to realize in today's vastly distributed world of cloud applications. Optimistic crash consistency [8] introduces a new API to separate ordering of file system operations from their persistence, enabling file system consistency in the face of crashes with asynchronous operations. Aerie [32] proposes a file system architecture that supports application-defined APIs for better efficiency to storage-class memory. We build on Aerie's approach and also support application-defined APIs, while avoiding problems with asynchrony by requiring synchronous operations.

NVM/Flash optimized file systems. Much recent work proposes specialized storage solutions for emerging non-volatile memory technologies. BPFS [9] is a file system for non-volatile memory that uses an optimized shadow-paging technique for crash consistency. PMFS [11] is a hardware/software co-designed file system that explores how to best exploit existing memory hardware to make efficient use of this type of byte-addressable memory. EXT4-DAX [6] extends the Linux ext4 file system to allow direct mapping of non-volatile memory, bypassing the buffer cache. NOVA [33] is a log-structured file system based on the EXT4-DAX framework. F2FS [18] is a log-structured file system design for flash memory that provides good performance on the medium. We take inspiration from these systems to provide a file system that can efficiently support multiple layers of diverse storage technologies simultaneously.

Multi-layer storage systems. These systems typically investigate tailored approaches to caching among different storage technologies. For example, RIPQ [31] is a novel caching algorithm that minimizes write amplification when moving data between DRAM and flash. FlashStore [10] is a key-value store designed to use flash as a fast cache between DRAM and HDD by minimizing the number of reads/writes done to the flash cache. Nitro [19] is a SSD caching system that uses data deduplication and compression to increase capacity. We expand these ideas to include support for efficient data migration among different storage layers.

NVM/Flash optimized applications/OSes. Researchers have also investigated optimizing both operating systems and applications for an underlying storage technology. For example, PASTE [16] proposes integrating network and storage stacks for faster persistence of network-initiated storage operations. Our approach provides application integration through customized APIs so that applications can be co-designed with the underlying storage technology.

6. Conclusion

Ongoing storage hardware trends are moving towards a multi-layer storage topology spanning multiple orders of magnitude in cost and performance. As these storage devices have become firmware-managed to hide the physical shortcomings of the storage media, hidden performance overheads can impact access performance and quality of service.

File systems that make use of this wide range of performance and capacity, while being conscious of the firmware management overheads, can provide a simpler interface to applications that delivers the best of all worlds—high performance, high capacity, and strong quality of service guarantees.

References

[1] Apache ZooKeeper. https://zookeeper.apache.org. [January, 2017].

[2] Elevator algorithm. https://en.wikipedia.org/wiki/Elevator_algorithm. [January, 2017].

[3] LevelDB. http://leveldb.org. [January, 2017].

[4] NVM Express 1.2.1. http://www.nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf. [January, 2017].

[5] SQLite. https://sqlite.org. [January, 2017].

[6] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/. [September, 2014].

[7] The Sprite Operating System. https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html. [January, 2017].

[8] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[10] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, Sept. 2010.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[12] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 307–320, New York, NY, USA, 2007. ACM.

[13] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

[14] T. Haynes and D. Noveck. Network file system (nfs) version 4 protocol, Mar. 2015. https://tools.ietf.org/html/rfc7530.

[15] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[16] M. Honda, L. Eggert, and D. Santry. Paste: Network stacks must integrate with nvmm abstractions. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 183–189, New York, NY, USA, 2016. ACM.

[17] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note*, 321211–002, Jan. 2011.

[18] C. Lee, D. Sim, J.-Y. Hwang, and S. 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.

[19] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 501–512, Berkeley, CA, USA, 2014. USENIX Association.

[20] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 147–160, New York, NY, USA, 2009. ACM.

[21] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[22] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, Aug. 1984.

[23] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[24] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). In *Acta Informatica*, 1996.

[25] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[26] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[27] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In

Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 433–448, Berkeley, CA, USA, 2014. USENIX Association.

[28] D. Shue and M. J. Freedman. From application requests to virtual iops: Provisioned key-value storage with libra. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 17:1–17:14, New York, NY, USA, 2014. ACM.

[29] C. E. Stevens, editor. *Information technology - Zoned Block Commands (ZBC)*. dpANS American National Standard, Jan. 2016. Project T10/BSR INCITS 536.

[30] I. Stoica. Challenges in big data (talk). In *Workshop on System Design for Cloud Services*, Redmond, WA, USA, July 2016. Microsoft Research.

[31] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[32] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[33] J. Xu and S. 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.