I/O in a Flash: Evolution of ONTAP to Low-Latency SSDs

Matthew Curtis-Maury, Ram Kesavan, Bharadwaj V R, Nikhil Mattankot, Vania Fang, Yash Trivedi, Kesari Mishra, and Qin Li, *NetApp, Inc*

https://www.usenix.org/conference/fast24/presentation/curtis-maury

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

I/O in a Flash: Evolution of ONTAP to Low-Latency SSDs

Matthew Curtis-Maury, Ram Kesavan*, Bharadwaj V R*, Nikhil Mattankot, Vania Fang,
Yash Trivedi, Kesari Mishra†, and Qin Li
NetApp, Inc

Abstract

Flash-based persistent storage media are capable of sub-millisecond latency I/O. However, a storage architecture optimized for spinning drives may contain software delays that make it impractical for use with such media. The NetApp® ONTAP® storage system was designed originally for spinning drives, and needed alterations before it was productized as an all-SSD system. In this paper, we focus on the changes made to the read I/O path over the last several years, which have been crucial to this transformation, and present them in chronological fashion together with the associated performance analyses.

1 Introduction

The advent of flash-based storage about a decade ago transformed the business of data center storage controllers. Despite improvements in several dimensions, the time to access any randomly selected data from storage had historically remained limited by physical constraints of spinning hard disk drive (HDD) technology. NAND-based solid state drives (SSDs) provided orders of magnitude lower latency and higher IOPS. In the last decade, several SSD-optimized or SSD-only architectures for data center storage controllers have been built and productized.

NetApp's® flagship feature-rich ONTAP® storage operating system is deployed in various configurations both within the data-center and in the cloud. ONTAP and its proprietary WAFL® file system [19] were optimized over their first two decades to maximize available I/O bandwidth on HDDs (with multi-millisecond latencies) for both reads and writes, and with a modular architecture to allow ongoing feature development. Most features of the WAFL architecture are required of any enterprise-quality storage system regardless of the underlying persistent media: data integrity [38], availability, data protection [51], recovery [28], etc. WAFL metadata was designed to optimize random metadata lookups from media, efficiently write out data and metadata to storage [25, 29], and to enable key functionality such as snapshots. Compression and deduplication techniques in WAFL improved efficiency in storage capacity. The WAFL consistency point converted random updates of user data and metadata into sequential I/O [13, 30], and wrote blocks to areas with the most free space, which turned out to be well-suited to minimizing FTL write amplification in SSDs.

ONTAP also integrated flash technology—PCIe-attached Flash Cache® [54] and SSD tiering in Flash Pool® [55]— but due to software delays in the ONTAP legacy data path, applications benefitted only partially from SSD's sub-millisecond latency, particularly for random reads. As such, ONTAP was faced with the challenge of making I/O software overhead commensurate with device latency in order to ship a competitive all-SSD system. Other legacy storage systems have similarly found software overhead out of proportion to low-latency device access times [8, 24, 32, 33, 40, 57, 62, 71]. Building a new storage architecture "from scratch" for SSDs would have required reinventing dozens of battle-tested features that were critical to our enterprise customers. As noted above, ONTAP and WAFL already had most of the building blocks necessary for building an all-SSD controller. Therefore, we instead reworked the legacy read path to speed it up incrementally over several software releases spanning multiple years, primarily by eliminating message hops between components of the storage software stack and moving towards a run-to-completion execution model that minimizes expensive message passing steps.

Although this paper focuses only on the optimization of the read I/O path, a collection of other improvements were also crucial to productizing the all-SSD controller. As described in prior work [25], we changed the block allocator to write contiguously down the SSD LBA-space in multiples of the SSD erase page size, thereby mitigating the log-on-log problem [67] and increasing SSD lifetimes. We redesigned the compression and deduplication infrastructure to run *inline* with writes, which reduces the overall data written to storage thereby further prolonging SSD lifetimes. We introduced other key performance optimizations, including in the write I/O path and journal replay.

This paper makes the following contributions: We analyze the latency breakdown of the legacy read path of a success-

*Currently employed at Google, †Currently employed at Meta

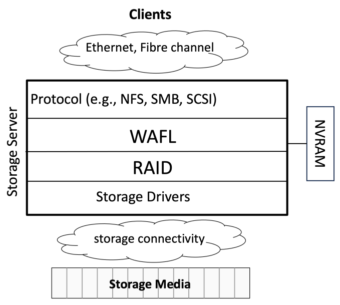


Figure 1: ONTAP modules involved in the data path.

ful enterprise storage system. We present a series of performance improvements made by systematically removing the primary sources of software delay. We analyze the improvements using data from detailed experiments across a range of hardware platforms and a cloud-resident VM. Finally, we discuss two major lessons that we learned from our experiences. Our improvements dropped software overhead from multiple milliseconds to less than 160us (more than 20X) generating large improvements in read latency and throughput, which has been foundational to the all-SSD ONTAP controller becoming a multi-billion dollar product line.

2 Background

In this section, we provide a brief overview of ONTAP and WAFL followed by a description and analysis of the legacy read path. We refer readers looking for a deeper understanding of WAFL to prior work [12, 13, 15, 19, 25, 26, 28, 29].

2.1 ONTAP Storage Stack

Fig. 1 shows the major ONTAP components in the data path. The Protocol component receives requests from clients and converts them into WAFL requests. The WAFL component processes all requests to the file systems—I/Os, operations related to data management, replication, etc. The layers beneath provide access to the storage media and implement RAID protection across them. Each component has data structures that are accessible typically only from thread pools dedicated to the component, which simplifies the synchronization between components. Component boundaries are traversed by message-passing between their threads, which means a request from a client undergoes multiple hops.

2.2 WAFL Processing Model

ONTAP houses and exports multiple file systems called volumes from within a shared pool of storage called an *aggregate*, and the WAFL component handles operations on them. The WAFL file system stores all metadata and user data in files which are organized in a hierarchical fashion. WAFL blocks are 4KiB in size and alignment, and are indexed in the aggregate by a PVBN (physical volume block number). Detailed descriptions are available elsewhere [15, 19].

Requests are dispatched to the WAFL component as WAFL messages. All data in the file systems are conceptually arranged into a hierarchy of data partitions called *affinities*, in a model referred to as *Waffinity* [12]. Based on its type and the data it intends to access, each request is dispatched to a specific affinity in the hierarchy. A dedicated pool of Waffinity threads execute requests on a per-affinity basis within the WAFL component in a thread-safe fashion. Each message type has an associated handler, which is coded in a *load-modify* transactional model: all resources necessary for the operation are accumulated in the *load* phase during which the message may suspend one or more times, after which the handler is completed in a single non-blocking *modify* phase, during which any mutations to the file system state are committed. This execution model together with the guarantees of Waffinity ensures that WAFL operations execute in atomic fashion with parallelism-safety.

If a resource is unavailable, the message releases all resources acquired thus far before it suspends (blocks) on an appropriate wait-list, thereby avoiding resource dependencies and deadlocks. When woken up, the message handler restarts execution from the beginning to try and reacquire the necessary resources. For example, a read message requires that the data blocks are available in memory. If those blocks are not in memory during the load phase, the read handler initiates retrieval of those blocks from persistent storage and suspends awaiting that retrieval. Upon restart, it goes through the same steps, but likely finds the blocks in memory this time and is able to complete its modify phase. This model provides deadlock-free concurrent execution but trades off CPU cycles for potential load phase re-execution.

2.3 Mutations to the File Systems

ONTAP was always designed to process mutations to the file system state with low latency. Consider the example of a write request. The load phase of the WAFL write message handler ensures the necessary inode and ancestor indirect blocks are in memory. The modify phase updates the file system state in memory and journals the write to NVRAM[1] before it responds to the Protocol component, which then sends an acknowledgement to the client. The modify phase leaves behind “dirty” file system state in memory—inodes, buffers, and volumes. Dirty state is collectively and periodically persisted on a per-aggregate basis as a single transaction called a *consistency point* (or *CP*) [13, 25]. Dirty data is compressed, deduplicated, and compacted [27] asynchronously to the write but before the CP completes. Because the inode

[1]ONTAP systems are deployed as HA-pairs, and the journaled write must get mirrored to the HA partner’s NVRAM as well.

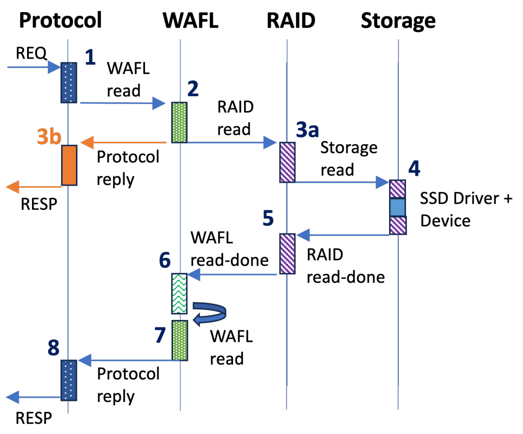


Figure 2: Message passing steps across ONTAP components for a read request.

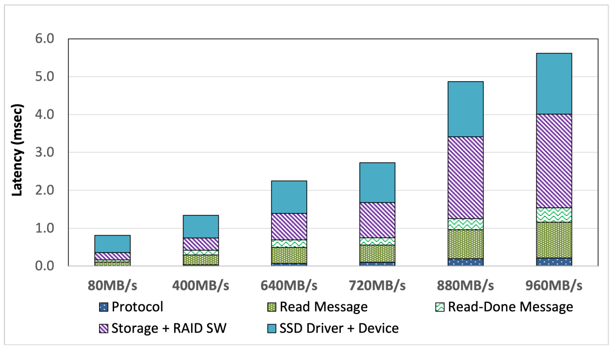


Figure 3: Latency across ONTAP components with increasing load for a 4KiB random read request.

and ancestor blocks of "hot" byte ranges are typically memory resident and appends to NVRAM are fast, the write latency is mostly independent of storage access times. Therefore, the read path was the main focus of the performance work required for productization of the all-SSD controller.

2.4 Legacy Read Path

Fig. 2 shows the message passing hops in the legacy read path and is applicable to all supported protocols—SCSI, NVMe, NFS, and SMB. In step 1, ONTAP receives the read request over the network in the context of a Protocol thread, which parses the request and translates it to a WAFL read message. A Waffinity thread picks up and executes the message in step 2. The read handler traverses file system data structures to find the requested data blocks. If the required data are found in memory during the load phase, the data is assembled into a reply payload in the modify phase and sent back to the Protocol thread, which replies to the client in step 3b. If not, the handler suspends until the data is available in memory. One such example is when all required intermediate data—inode, indirect blocks, etc.—are found, but the data blocks are not. In this case, the handler allocates, initializes, and inserts one buffer per missing block in the file system tree, places them in one or more RAID read messages that it sends to the RAID component, and suspends the WAFL read message on the completion of all required I/Os. WAFL uses the PVBNs of the blocks to determine the required number of RAID read messages.

A RAID thread processes each RAID read message, uses its knowledge of the drive mappings to translate each PVBN to drive ID and LBA, and sends a message to the Storage component in step 3a. In step 4, a Storage thread processes this message, dispatches a read to the physical drive, and sends a read-done to the RAID component upon completion of the I/O. In step 5, RAID validates checksums and sends a read-done to the WAFL component if no errors are found. In step 6, a WAFL thread does further validation[2], marks the buffers *valid*, and restarts all waiters. The original WAFL read message is awoken once all issued RAID messages (from step 2) have completed. In step 7, a Waffinity thread runs the original message by re-executing the handler, eventually finds all valid buffers in cache, and replies to the Protocol component, which replies to the client in step 8.

Three different data reduction techniques—compression, deduplication, and sub-block compaction [27]—are used in combination by WAFL to efficiently store data; the data is also encrypted just before it is stored. Decryption occurs in step 4 while reading from storage, but the choice of where (in one of the steps in the reply path) the reduced data gets rehydrated is made dynamically based on various conditions. We consider the topic of data reduction outside the scope of this paper for two reasons: it is too large a topic to cover comprehensively, and it would be a distraction because the techniques and results presented in this paper are fundamentally unchanged with or without data reduction.

This architecture is modular, which facilitates continuous feature development, and error handling can be performed in the corresponding component. A WAFL read that does not hit in the buffer cache may incur several message hops including multiple suspensions and restarts within WAFL before completion. Such hops become expensive under CPU pressure, when a message must wait for the next thread to be scheduled or when running threads cannot keep up with incoming load. In the case of HDDs, these scheduling delays are typically dwarfed by drive I/O latencies. Such delays become noticeably large for SSDs.

2.5 Components of Read Latency

Fig. 3 shows the breakdown of *server-side* latency for a read request to ONTAP across the steps outlined in Fig. 2 under

[2]WAFL stores a *context* together with each written block [60] to identify its file and offset to protect against lost or misdirected writes [3], and identify a block that has been moved for defragmentation purposes [26].

increasing levels of load, using a matching color scheme for each step. This data was collected on ONTAP 8.2.2 (circa 2014), which predates the optimizations discussed in this paper. A random read workload—which ensures a low buffer cache hit rate and frequent drive access—was run on a 2x10-core Intel Xeon 2.8GHz controller with 128 GiB of DRAM, the high-end ONTAP system from that time. The controller had twelve 400GiB SAS SSD drives, which collectively provided sufficient I/O throughput for the highest load of this experiment. A set of LUNs were configured on ONTAP and a number of clients sent 4KiB reads to random offsets using the FCP storage protocol over an underlying Fibre Channel network to cumulatively create the desired load. Throughout this paper, time within each component is measured using start/stop timers in the software stack. Network component time is included within the Protocol layer, which collectively remain a small source of delay due to their relative efficiency compared to other components in the stack.

Protocol corresponds to steps 1 and 8. *Read Message* and *Read-Done Message* depict time in the WAFL component, the former for the sum of steps 2 and 7 and the latter for step 6. *Storage+RAID SW* corresponds to steps 3, 4, and 5 minus *SSD Driver+Device*, which depicts the latency in the device driver and media. The raw CPU cycles in the WAFL and Protocol components (steps 1, 2, 6, 7, and 8) are negligible (each less than 30us); in other words, most of the latency is the message waiting to be picked up by WAFL or Protocol threads. Although *SSD Driver+Device* time does increase with load, all of that increase is attributed to software delays in the device drivers due to increased CPU wait times. We confirm this later in Fig. 7, which shows consistent *SSD Driver+Device* times when CPU wait times are not a major factor. Increased load amplifies the cost of each hop because threads are busier and CPUs are closer to saturation.

At 80% of maximum throughput of the system (960MiB/s), the device latency is less than 30% of the total read latency. The primary non-device delays are in the RAID/Storage components and wait times in WAFL for the read and read-done messages. While such delays were acceptable for HDDs with media latency of several milliseconds, their impact became outsized for SSDs with media latency of a few 100's of microseconds. Clean-sheet design approaches for the read path were discarded because of the inherent complexities around handling myriad error conditions and integrating with existing ONTAP features. Instead, we used the latency data to iteratively improve the read path for the most common cases while retaining the legacy path for error handling and other complicated conditions.

3 Fastpaths: WAFL Reply and RAID

Optimization of the read path for SSDs started as a skunkworks project in the WAFL team, and we began with the WAFL reply path. Because the latency breakdown was dominated by wait-times due to message passing hops, we chose to eliminate hops, steps 6 and 7, instead of optimizing code. We call this work *WAFL Reply Fastpath*. Next, the RAID and Storage teams eliminated steps 3a and 5, called *RAID Fastpath*. These changes are shown in Fig. 4.

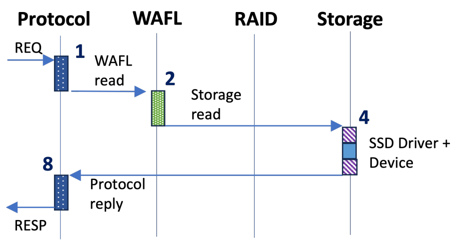


Figure 4: Steps with WAFL Reply and Storage Fastpaths.

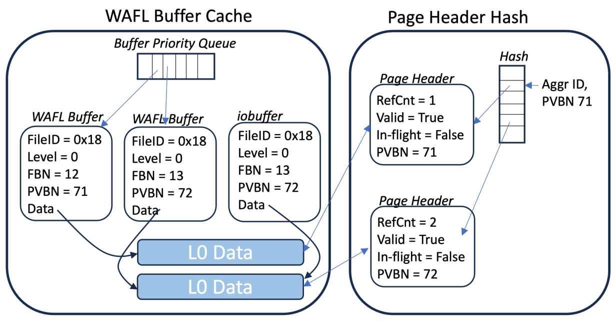


Figure 5: WAFL buffer cache and page header hash.

3.1 Bypassing WAFL Read-done

The WAFL read-done message validates the data, updates the WAFL buffer state to reflect the I/O completion, and restarts the original WAFL read message. We explored whether the error-free path of this handler could be executed directly by the RAID component as part of RAID read-done (step 5). We refer to this technique as *bypassing layers*. Data blocks and indirect blocks of a file are represented in memory as *WAFL buffers*, which are logical headers that point to 4KiB *data block pages*. WAFL buffers are arranged into per-file inode block trees. A multi-level LRU structure [14], labeled Priority Queue in Fig. 5, tracks the aging and priority of buffers, and is designed to be accessible from within and outside of the WAFL component as a result of earlier performance improvement work.

A given data block can be shared by several inodes' block trees; this capability is used by many ONTAP features, such as snapshots, deduplication, cloning, etc. Hence, multiple WAFL buffers can point to a data block page. Each block page also has a statically-associated *page header*, that stores metadata about the page such as a reference count of the WAFL buffers pointing to it. The page headers are tracked in a header hash indexed by aggregate ID and PVBN, which is looked up before issuing I/O. Fig. 5 shows two example block pages with PVBNs 71 and 72. Access to block pages and page headers are protected by range locks on the page header hash from any component. A block page can only be scavenged when its page header refcount is zero, which implies all buffers pointing to it have been evicted.

In the legacy read path, a WAFL buffer (per block) was sent with the RAID read message. The RAID and Storage components could safely update certain flags/fields in those buffers to track I/O state, error states, the checksum, etc. However, marking the buffer valid could happen only within the WAFL component, hence the need for WAFL read-done. In the new model, we add a valid state in the page header and leverage a new *iobuffer* object that is used exclusively for the purpose of I/O and is therefore exempt from many of the rules that govern WAFL buffers. As in the legacy path, the WAFL read message inserts a WAFL buffer but now also initializes an iobuffer per block, which it instead sends with the RAID read message. The iobuffer is private to the read request and cannot be found otherwise. Both buffers point to the same block page, as shown in Fig. 5. The WAFL read message now suspends on a page header (instead of a WAFL buffer) waiting for it to become valid. The RAID read-done message first validates the data block then directly invokes a WAFL function that performs the file system specific validation, marks the page header valid, wakes up the suspended WAFL read message, and frees the iobuffer. If it encounters any errors, it can safely fail through to WAFL at any point, because WAFL messages always restart execution from the beginning of the message handler.

3.2 Bypassing the Restart of WAFL Read

The removal of step 6 resulted in significant improvements, and encouraged us to next explore eliminating step 7. In the legacy read path, the restarted WAFL read message ensured that all data was present in memory, assembled them into a vector, and replied to the Protocol component. As with the WAFL read-done message, this work is now executed inline by the RAID read-done (step 5) message by using iobuffers. The original WAFL read message is attached to the RAID read message, in which we keep count of the outstanding I/Os to storage. This count is atomically decremented upon each I/O completion, and the last completion replies to the Protocol component. If any errors are encountered, the legacy path is triggered by sending the read message back into WAFL. When a Protocol thread receives the reply, the embedded WAFL read message is freed. The original WAFL buffer is marked valid only if accessed by some subsequent WAFL message (or the restarted WAFL read message in case of an error) on finding that the buffer points to a valid block page. If never accessed, the buffer eventually ages out like any other. The elimination of steps 6 and 7 is collectively called the WAFL Reply Fastpath.

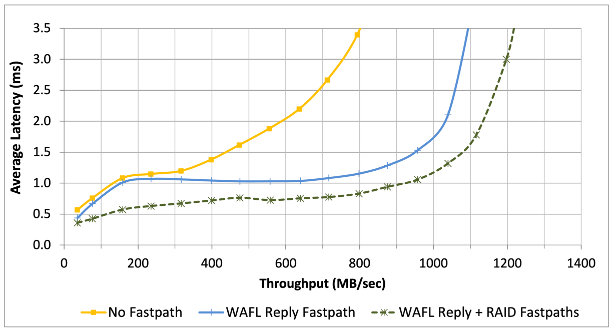


Figure 6: Latency vs achieved throughput with increasing 4KiB random read load with and without Fastpaths.

3.3 RAID Fastpath

We next worked to bypass the RAID component (steps 3a and 5) entirely on the read path. The RAID component maintains an up-to-date topology data structure of the aggregate; it knows which drives are in some failure state or are getting reconstructed. RAID uses that information in RAID read (step 3a) to map the PVBN of each buffer supplied by WAFL to the physical drive and LBA. RAID exports a read-only cache of the topology, which is now used by the WAFL read message for the translations and to directly send I/O messages to the Storage component. Changes in the aggregate, such as addition of drives, failure of drives, or RAID reconstruction, will require updating the topology. Though rare, when such events occur the RAID component flags the cache as stale, and the WAFL read message fails through to using the legacy RAID read. In the case of a race—say the topology is tagged stale after a Fastpath is triggered—the Storage component detects the staleness in step 4 and returns an error, and the restarted WAFL read message now fails through to the legacy path. The Fastpath resumes once a new topology cache has been built and exported by RAID.

Upon completion of a device I/O in step 4, the Storage component now directly calls a thread-safe version of the checksum validation used in RAID read-done (step 5), followed by the WAFL Reply Fastpath described in Sec. 3.1 and Sec. 3.2. As elsewhere, the legacy path is used as a fail-safe whenever any error is encountered.

3.4 Performance Analysis of Fastpaths

Fig. 6 and 7 show results from the same 4KiB random read experiment on the same 20-core platform from Sec. 2.5 with the Fastpaths enabled. *No Fastpath* data was collected by using ONTAP 8.2.2 (circa 2014), which precedes our optimizations, *WAFL Reply Fastpath* using ONTAP 8.3.0 (early 2015), and then with *RAID Fastpath* using ONTAP 8.3.1 (late 2015). Although not strictly apple-to-apples because we are comparing different releases, the performance impact seen in these graphs is primarily due to the Fastpaths. Fig. 6

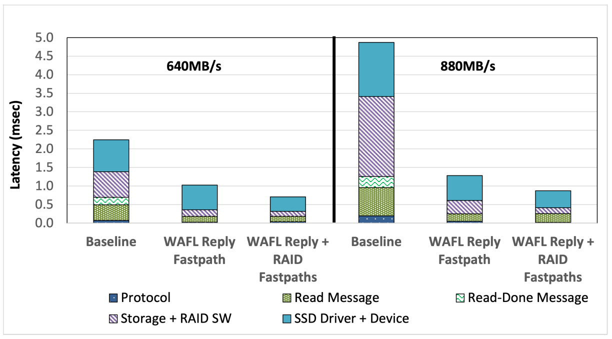


Figure 7: Latency across ONTAP components with and without improvements at two specific loads of 4KiB random reads.

plots the average server-side latency vs achieved load, and shows how these improvements have significantly shifted the system saturation points to the right. The read throughput at the average latency of around 1ms (the industry expectation for SSD-controller latency in the mid-2010's) quadrupled from 150MiB/s to 600MiB/s with WAFL Reply Fastpath, and increased another 50% from 600MiB/s to 900MiB/s with RAID Fastpath. Fig. 7 shows the latency breakdown at two specific load points. The sharp increase in latency with the legacy path is attributable primarily to the RAID and Storage components. From mining finer grained statistics in ONTAP, the savings at 880 MiB/s compute to 300us of wait time for the WAFL read-done message, 480us wait time for the restarted WAFL read message, and a smaller 17us of CPU time across both messages. More interestingly, the reduction in CPU utilization due to the elimination of steps 6 and 7 results in lowered wait times for threads in all components, lowering the overhead of remaining message hops and deferring CPU saturation to higher levels of load. Adding the RAID Fastpath at 880 MiB/s results in a further reduction in wait times in RAID and Storage components and a reduction in device driver wait time. In the end, software overhead is on par with device times.

It should be noted that latency variance in ONTAP is almost always due to variance in wait times, which gets worse only with increased CPU utilization. Therefore, latency variance is high only to the right of the "knee" of the latency-throughput curve [50]. Because Fastpaths (and the improvements presented subsequently in this paper) significantly reduce wait times, their benefits for p90 and p99 latencies have an outsized impact to the right of the knee of the curve. For example, p90 latency drops from 7ms (for legacy) to 2ms (with both Fastpaths) in this experiment. Therefore, we use average latency as a conservative showcase of the improvements throughout this paper.

In this section, we presented and evaluated a collection of optimizations to minimize message hops in the read I/O path. We showed that component layers can be effectively bypassed by running limited elements of one layer within

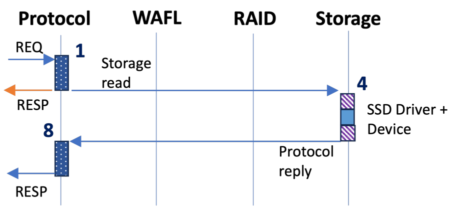


Figure 8: Read path steps with TopSpin read design.

another layer to constrain software overhead. This work was crucial to NetApp shipping a feature-rich all-SSD ONTAP controller in 2015 instead of creating an SSD-optimized file system from scratch. Further, the success from Fastpaths encouraged the continued use of this approach in ONTAP. The next section discusses how we used bypassing to tackle the dominant remaining delay.

4 TopSpin Read: Bypassing WAFL Read

By the early 2010s, it was obvious that traditional interconnects such as SAS and SATA were inadequate for SSD speed and bandwidth. Based on the new NVM Express technology [52], enterprise-quality SSDs boasting at least one order of magnitude better performance were available by the late 2010s. NVMe storage drivers were added to ONTAP to access these new SSDs. Additionally, Linux and Windows clients were now able to unlock these performance benefits by connecting over the network using the NVMe over Fabrics (NVMe-oF) protocol. In response, an NVMe server-side module optimized for parallelism and low latency was added to the ONTAP Protocol component. With these technological improvements, tackling the remaining large software delay—the wait time for WAFL Read (step 2) as seen in Fig. 7—became a competitive imperative.

To that end, we developed *TopSpin*, an optimization to allow the common-case read request to bypass the WAFL component, as shown in Fig. 8. TopSpin leverages direct access to WAFL data structures from within the Protocol component (step 1) to check if the required data is in memory and to issue I/Os to storage, while handling all potential races with requests that modify file system state running in parallel within the WAFL component. This design has three advantages: (1) It avoids the queueing delays within WAFL. (2) The reimplemented read handler is light-weight and avoids the suspend-restart CPU overhead. (3) It bypasses the strict data partitioning within WAFL that can restrict parallelism. It was productized in ONTAP 9.3 (2017) and enabled for all block-based protocols—SCSI and NVMe.

4.1 Storage Location Cache

We introduce the Storage Location Cache (SLC) to allow the Protocol component to directly and safely discover data lo-

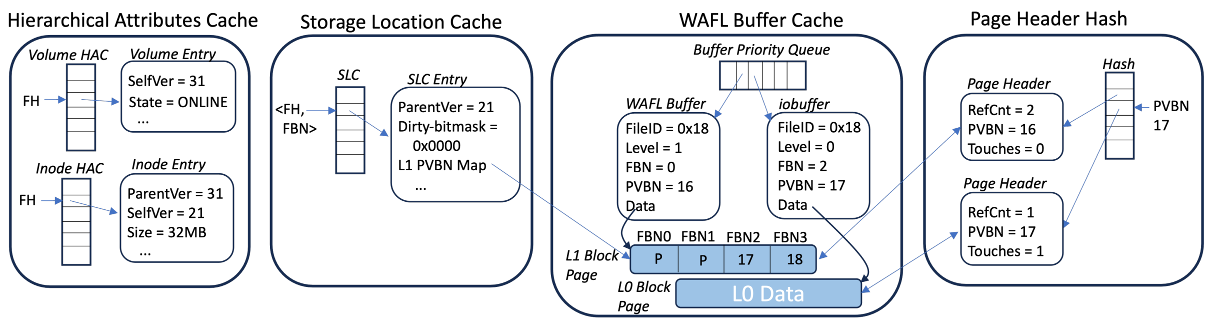


Figure 9: Overall SLC and HAC architecture, which integrates with the existing WAFL buffer cache and page header hash.

cations. The SLC is a hash table that maps file handle and file block number (or FBN, the 4KiB file offset) to PVBN; its hash buckets are protected by range locks. The lowest level indirect blocks in a WAFL inode tree (Level-1 blocks, or *L1s*) comprise this map, along with per-FBN auxiliary information used for data validation. The location of the i^{th} FBN is found in the $(i\%span)^{th}$ index of the $(\frac{i}{span})^{th}$ L1, where the fixed span is the maximum number of children an L1 can have. As Fig. 9 shows, each SLC entry points directly to a block page of one L1 and the SLC entry takes a refcount through the corresponding page header. SLC entries are inserted (when an L1 block page is loaded into memory) and updated only from the WAFL component, including being removed when the L1 page is scavenged.

4.2 Hierarchical Attributes Cache

For a read request to be safely processed in the Protocol component, it must synchronize with changes to file system state occurring in parallel within the WAFL component. For instance, changes to the mount state of a volume or the size of a file may interact with a read request. We introduce the Hierarchical Attributes Cache (HAC) to track properties of file system objects—inodes and volumes—to facilitate such checks. Each user file or LUN is represented by an HAC inode object that caches various attributes of the inode, such as size and permissions. Each volume is represented by an HAC volume object that caches mount state, encryption key, etc. As Fig. 9 shows, the objects are organized into two hash tables indexable by file handle (which includes a volume identifier). Access to these objects is protected by a lock per hash bucket. Much like the SLC, HAC objects are consulted from the Protocol component but created and updated only from the WAFL component; a volume (inode) object is added to the HAC when it is mounted (loaded into memory).

4.3 TopSpin Read from Protocol Component

We implement *TopSpin read*, a version of the WAFL read handler that is called directly by the Protocol thread towards the end of step 1. Fig. 9 shows the system state for an example TopSpin read of FBN2 and FBN3 of a file. It first looks up the SLC using file handle and offset to determine if the requisite L1 block pages are in memory; the actual lookup converts the offset to the FBN aligned to L1 span, which is FBN0 in this case. If found, it confirms the freshness of the SLC entries by consulting the HAC inode and volume objects; Sec. 4.5 details the the freshness check. Next, it indexes the L1 page to obtain the PVBNs (and auxiliary information), 17 and 18 in the figure, and looks them up along with aggregate ID in the page header hash. If all block pages are found in memory, it inserts them into the reply vector and replies to the client, holding a page refcount until complete. Otherwise, much like the WAFL read handler, TopSpin uses the PVBN and auxiliary information to instantiate iobuffers, block pages, and page headers, and sends the appropriate I/Os to the Storage component for the missing Level-0 file data blocks (or *L0s*). The Protocol component resumes processing this request after receiving a reply from Storage, much as in Sec. 3.3. If TopSpin read fails for any reason, such as missing L1 block pages or freshness check failure, it falls through to the legacy WAFL path. Both caches—HAC and SLC—use LRUs to age their entries, thereby increasing the chances of TopSpin reads to "hot" file byte ranges completing successfully.

As noted in the Fastpath sections, some data structures were already safe to access from outside the WAFL component—the block pages, page headers and hash, the RAID topology cache, etc. TopSpin limits itself to accessing only those shared structures. When an I/O completes, step 4 now inserts the iobuffer directly into the buffer cache LRU, unlike in the Fastpath case where the iobuffer is discarded and the WAFL buffer is preserved. TopSpin reads access L0 block pages directly through the page header hash, and increment a newly added *touches* count field in the page header to track hotness (shown in Fig. 9). Before an iobuffer can be scavenged, any such references are transferred from the corresponding page header into the iobuffer thereby preventing eviction. We next look at how the SLC and HAC guarantee correctness.

4.4 Keeping SLC Consistent

A write request executing in parallel within the WAFL component may conflict with a TopSpin read, and we use the SLC entry for synchronization. Each SLC entry uses a *dirty-bitmask* to track whether its children data blocks have been "dirtied" by any operation running in the WAFL component, one bit per child. In its modify phase, the WAFL write handler locks up to 3 SLC entries—the largest write supported (1 MiB) may span up to 3 L1s—to set the dirty bit for each FBN. A TopSpin read looks up the dirty bits after locking the necessary SLC entries, and fails through to WAFL if any is set. Otherwise, it obtains the PVBNs and either finds the block pages through the page header hash or issues I/Os.

The subsequent CP walks through each dirty buffer and allocates a new location for it in storage, a previously free PVBN. Then, it rehashes the dirty block page using the new PVBN in the page header hash, after which the page is sent together with several other pages as a write I/O to a RAID group in the aggregate; more details in other work [13, 25]. In theory, each dirty bit in an SLC entry can be cleared when the CP rehashes the child L0 block page with the new PVBN—a subsequent TopSpin read can now safely read that block page. Instead, to amortize locking costs, all dirty bits in an SLC entry are cleared together by locking the SLC entry just once after the CP is done with the L1 and all its children. It typically takes anywhere from 2-5 seconds for a subsequent CP to process that parent L1 and clear the dirty bits in the SLC entry. This is rarely a problem for our customer environments, where immediate reads after writes are rare, but would result in failing through to WAFL.

4.5 Keeping HAC Consistent

SLC entries may be invalidated by various infrequently occurring operations, such as a volume remount or a file resize. These are tracked by versioning HAC objects. Each HAC object records two version numbers: a self version v_s for its child relationships and a parent version v_p for its parent relationship; Fig. 9 refers to them as SelfVer and ParentVer, respectively. The former is initialized when an object is created and incremented when any of its attributes change. For example, if a file is resized its inode object's v_s gets incremented. Each SLC entry also records a v_p. A hierarchy exists: each SLC entry has a parent inode, and each inode HAC object has a parent volume HAC object. When an HAC object is created (updated), its v_p is initialized to its parent's v_s and its own v_s is set (incremented). An object or entry is confirmed to be fresh only if its v_p matches its parent object's v_s. A check must recurse up the hierarchy to confirm freshness.

A failed SLC entry check at the inode (or volume level) indicates that the corresponding file (or volume) has since been modified in some way that makes the SLC entry stale. When that happens, TopSpin sends the read to the WAFL component. Stale SLC entries and HAC objects age out of their respective caches. Version numbers are incremented only by operations running within the WAFL component. Incrementing the version of an object implicitly invalidates all its descendent objects, which may be numerous—a volume may comprise hundreds of files, each with thousands of "hot" L1s. It should be noted that version bumps occur infrequently, and therefore the fast 3-level recursive check done by a TopSpin read succeeds most of the time.

4.6 TopSpin and File-based Protocols

The improvements in Sec. 3 moved portions of the read path from WAFL and RAID down to the Storage component, and are therefore independent of the client protocol. All protocols can benefit from the Fastpaths. In contrast, TopSpin read requires changes to the code in the Protocol component. We implemented TopSpin first for block-based (SAN) protocols because SAN applications—such as databases, server virtualization, and business applications—require consistently low latency. NAS protocols require that a read check other metadata, such as file permissions, ACLs, and lock state. These structures are currently accessible only from within the WAFL component. An inode's access time (*atime*) also needs to be updated on reads, which results in mutations that need to be persisted. A TopSpin read would need to safely access the corresponding structures.

In this section, we extended Fastpath to bypass the WAFL layer in the read I/O path, by developing an alternative method for scalable, thread-safe file system accesses. Thus far, it has been narrowly deployed within ONTAP, but it can be applied to other protocols and file system operations. Extending TopSpin to NFS and SMB is a work in progress.

5 Client-Visible Consistency Semantics

In the previous sections, we discussed correctness in the face of race conditions within and between components. We now look at correctness from the client's point of view. Clients communicate with ONTAP using one of several protocols—NFS, SMB, SCSI, and NVMe-oF—each with its corresponding correctness semantics. To maximize code reuse and to simplify design and testing, ONTAP implements a conservative and consistent interpretation of the semantics across all protocols. Changing these interpretations is disallowed across ONTAP releases because it risks destabilizing client libraries and customer applications. All improvements presented in this paper preserve ONTAP's interpretations of these semantics. We look at only two rules that are relevant to this paper. In this section, we use the term "write" generically for any operation that mutates file system state and "read' for any that does not.

For example, if a client issues a read R after it has received the acknowledgement to a write W, then R must never see any file system state prior to W. Because a server cannot know the exact moment when an acknowledgement is received by a client, we implement rules based on when requests (acknowledgements) enter (exit) the networking stack of the Protocol component. (1) *Read-After-Write* (RAW): ONTAP guarantees that once the Protocol component has issued an acknowledgement of a write, a read request received subsequently by the Protocol component sees only the state after the write. (2) *Concurrent-Read-Write* (CRW): If a read and write overlap in time when processed by ONTAP, the read sees state only from before or after the write, but never both; except for SCSI, where CRW applies only for sizes up to 64 KiB[3]. Both rules hold even if the read and write are sent by different clients using different protocols. In legacy ONTAP, every file system request was processed by the WAFL component in both the request and reply paths. The load-modify transactional model together with Waffinity guaranteed serialization of a read and a write if they conflicted in file byte range; this trivially satisfied both rules. The improvements presented in this paper are relevant only to read requests—one of the many possible "reads" as used generically in this section.

The Fastpaths avoid the RAID and WAFL components on only the reply path, so trivially preserve RAW. Because the results of a write are committed to the in-memory file system state by the WAFL component before its acknowledgement can be sent, it is impossible for a TopSpin read to see content from prior to the write. Thus, TopSpin also preserves RAW.

With both Fastpaths and TopSpin, if the write runs first then the read sees data from only after the write. In the case of TopSpin, the write first locks all SLC entries and sets the dirty bits in them, so the read fails the freshness check and falls through to WAFL. If the read runs first and finds all data in memory, it replies with data from only before the write in both Fastpaths and TopSpin. The TopSpin read locks all the SLC entries it needs. If the read runs first and finds that all its L0 pages are missing, it uses PVBNs from the L1s to issue I/Os to the Storage component. TopSpin finds the PVBNs via the SLC entries and the WAFL read handler via the L1 buffers. Even if a subsequent write dirties one or more of those FBNs, the read replies with only the persisted data prior to the write. Any read with a mix of hits and misses in the page header hash fails through to the legacy path. Although this case can be improved, it does not occur often in our customers' applications. Thus, CRW is preserved.

6 Performance Evaluation of TopSpin

A typical ONTAP controller hosts datasets for multiple instances of different applications that are accessed at the same time. No individual workload represents all outcomes in such multi-tenant environments. Therefore, we primarily used micro-benchmarks to study the performance, knowing that the results extend to any workload comprising those traffic patterns. We also tested with an in-house benchmark identical to the industry-standard SPC-1 [11], which models the query and update operations of an OLTP/DB application and simulates real world environments [17]. Lastly, we used a standardized load to Oracle. All experiments used a recent internal build based of ONTAP 9.13.1, unless otherwise indicated. We picked a *mid-range controller* with 2.2GHz Intel Xeon Silver 2x10 cores, 144GiB of DRAM, 16GiB of NVRAM, and 23 3.84TiB NVMe SSD drives to study the benefits when CPU resources are tight and a *high-end controller* with 2.2GHz Intel Xeon Platinum 2x32 cores, 1TiB of DRAM, 64GiB of NVRAM, and 46 3.84TIB NVMe SSD drives. The NVMe SSD drives support 100K IOPS of random reads with latency under 100us, and are configured into RAID double parity [10]. These configurations are realistic and are sufficient to make most workloads CPU-limited. A given IOPS load is collectively initiated in an open loop by a set of remote clients, such that queuing in the server becomes significant under heavy load. Latency is measured on the ONTAP server from when a read request enters to when its corresponding reply exits the controller. The TopSpin SLC is backed by L1 block pages in the WAFL buffer cache which can consume the majority of a system's DRAM and prioritizes indirect blocks. An L1 page in WAFL can reference 255 child blocks, so TopSpin can be effective with sizes significantly smaller than an application's working set.

Controllers are deployed as a high-availability pair, but we report only per-controller results. Only half of NVRAM is used by a controller because the other half is used to mirror the HA-partner's journal. Although compression, deduplication, and compaction [27] are now enabled by default on ONTAP all-SSD controllers, we disabled them in these experiments for three reasons: (1) Enabling them on datasets with realistic compressibility and dedupe savings does not change the character of the results. (2) We have not presented the designs of these data reduction techniques and how they interact with the read path. (3) We lack the space to explore the range of datasets that yield different combinations of compressibility and dedupe savings. Available CPU cycles in all-SSD systems can be used for running data reduction, both inline and in the background. Thus, savings in CPU cycles can directly benefit storage efficiency.

6.1 Reads: NVMe-oF Clients

In the first experiment, the load-generating clients used the NVMe-oF protocol (over a Fibre Channel network) to communicate with LUNs configured on ONTAP. Together with NVMe SSDs, when compared to earlier results, the latency and throughput numbers are both an order of magnitude better. At these low latencies, the experiment is more sensitive

[3]The SCSI specification does not require atomicity. ONTAP does not support WRITE_ATOMIC.

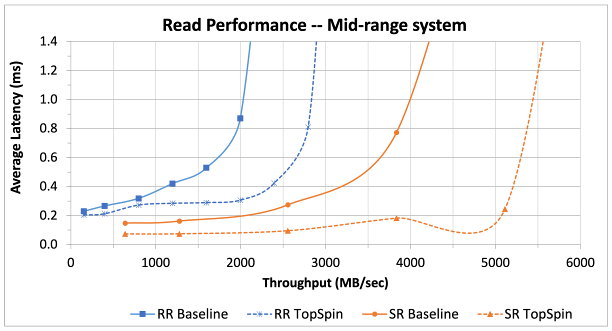


Figure 10: Latency vs achieved throughput on the 20-core system with increasing random read (RR) and sequential read (SR) load.

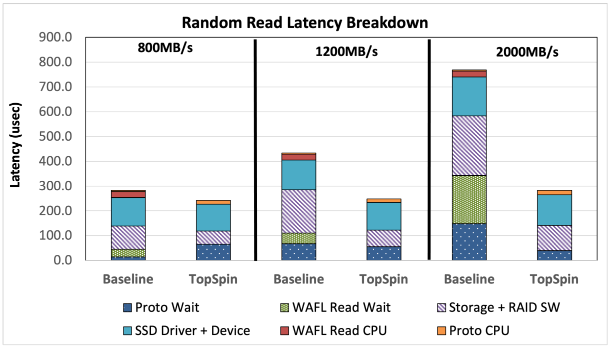


Figure 11: Latency across ONTAP components with and without TopSpin at three specific loads of random reads.

to software delays in ONTAP. Fig. 10 presents the latency vs achieved throughput on the mid-range 20-core controller with 8KiB random read (*RR*) [4] and 64KiB sequential read (*SR*) workloads. *Baseline* now includes the Fastpaths.

6.1.1 Random Read Performance

TopSpin shifts the curve to the right, for example doubling throughput at 400us latency. Customers can also operate their systems for higher throughput, with a tolerance for higher latency (e.g., 5ms). The *peak* throughput of a system is that achieved as the system approaches saturation and beyond which latencies grow exponentially. As with the Fastpaths, TopSpin delivers a 27% higher peak throughput because the streamlined I/O path reduces the CPU costs per operation (3.0GiB/s at 2.9ms latency vs. 2.4GiB/s at 3.2ms). In this test, TopSpin finds the required data in memory in 1.9% of reads, issues storage I/O directly in 97.7% of reads, and falls through to WAFL in only 0.4% of reads for reasons such as an unavailable SLC entry. Fig. 11 shows the per-component latency at three loads, and WAFL and Protocol latencies are now further divided into CPU time vs wait time. The data at low load (800MiB/s) approximates the break-

[4] By this time, the official SPC-1 as well as our internal benchmarks had switched the I/O size from 4KiB to 8KiB for random I/O workloads.

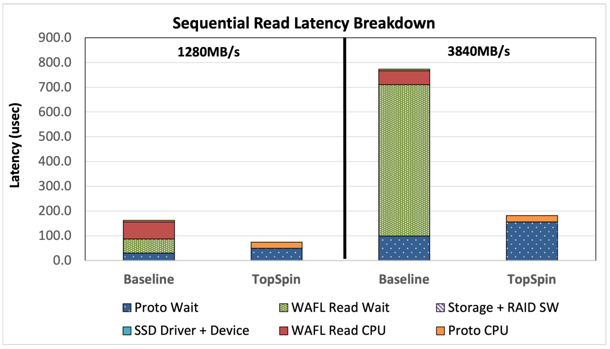


Figure 12: Latency across ONTAP components with and without TopSpin at two specific loads of sequential reads.

down for a single outstanding I/O. As load increases, the WAFL read message wait time becomes a significant factor (195us at 2GiB/s), and TopSpin eliminates it. Reduction in CPU consumption also yields lower wait times in other components. As we are now evaluating with NVMe SSDs, *SSD Driver+Device* latency ranges from 115us to 160us without TopSpin, compared to older generation SAS SSDs from Fig. 7. With TopSpin, this drops to 108us to 123us, due to decreases in driver scheduling delay. TopSpin replaces 24us of WAFL read message CPU time with a 10us increase in the *Proto CPU* time, which results in CPU cost per read dropping from 66us down to 52us. At 2GiB/s, the non-device related time drops from 690us to 160us, in better proportion with the 123us *SSD Driver+Device*. *With TopSpin, device latency is now the largest single component and non-device latencies remain below 60% of the total.*

6.1.2 Sequential Read Performance

In general, ONTAP is capable of much higher sequential read throughput because it uses speculative readahead to prefetch required data into memory. This effectively eliminates *SSD Driver+Device* from the latency path, as shown in Fig. 12. As expected, TopSpin eliminates *WAFL Read Wait* and replaces substantial *WAFL Read CPU* time (56us at 3840MiB/s) with a small increase in *Proto CPU* (20us), resulting in significant increase in throughput—e.g., a 40% increase at 800us latency. Peak throughput goes up by 19% (6.0GiB/s at 3.0ms vs. 5.0GiB/s at 3.2ms), due to a reduction in the per operation CPU cost from 251us to 209us. Thanks to readahead, TopSpin finds data in memory in over 99.99% of reads and avoids failing through to WAFL.

Unlike TopSpin random reads, readahead prefetching instantiates and inserts WAFL buffers. We next optimized the readahead engine to use iobuffers. This carries two benefits: (1) iobuffers are lighter-weight because they maintain less state and so access fewer cache lines, which reduces the CPU cost of processing both their insertion and eventual eviction. At 5.1GiB/s load, the average CPU cost

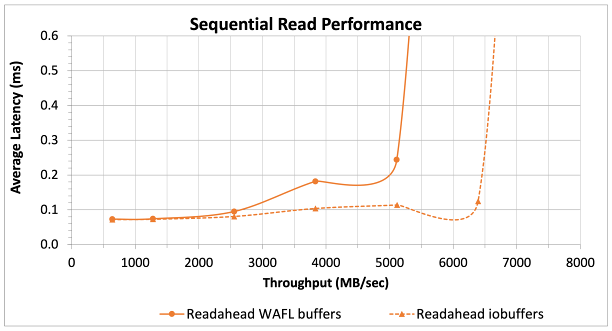


Figure 13: Latency vs achieved throughput on the 20-core system with TopSpin on sequential read load, using WAFL buffers and iobuffers for readhead.

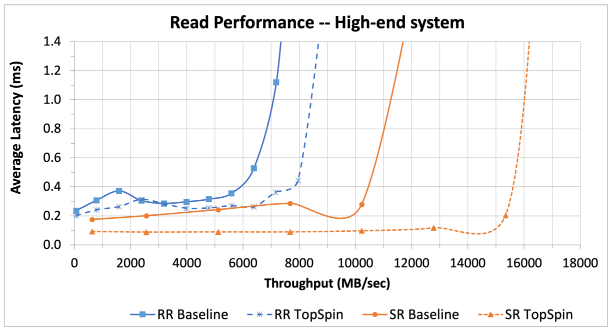
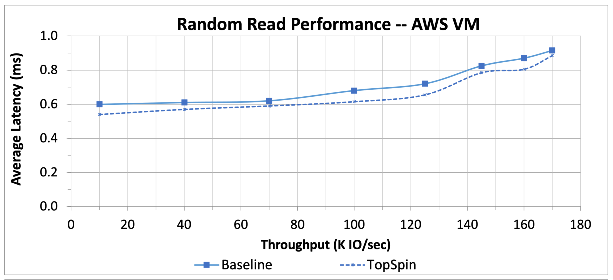


Figure 14: Latency vs achieved throughput on the 64-core system with increasing random read (RR) and sequential read (SR) load.

of readahead drops from 24.5% of all cores to 16.5%, and buffer scavenging drops from 15.0% of all cores to 6.0%. (2) iobuffers can be scavenged from outside of the WAFL component, which helps reduce overall WAFL wait times; none of that 6.0% scavenging CPU cost is in the WAFL component. Fig. 13 shows the results of this approach, including a 19% increase in peak throughput (7.2GiB/s at 2.3ms vs. 6.0GiB/s at 3.0ms).

6.1.3 High-end Read Performance

Fig. 14 reports the results of the same random and sequential read experiments on the high-end 64-core controller to study TopSpin on controllers with more CPU cores. Compared to the 20-core system, TopSpin benefits are similar for SR but are somewhat lower for RR. This shows that TopSpin benefits are greater for certain workloads when CPU resources are more limited. The latency bump around 2GiB/s for both RR graphs is due to the time-lag to activate the optimal number of threads in the (NVMe) Protocol component on this high-end controller; as mentioned earlier, ONTAP dynamically scales this number. The per-component latency breakdown (not shown) matches that of the 20-core controller.



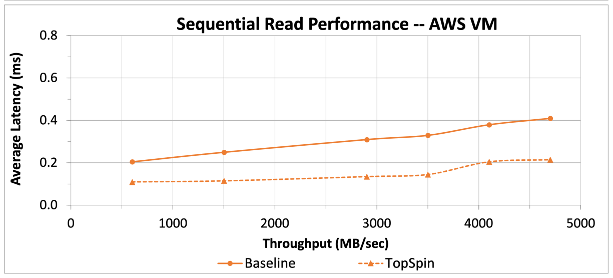


Figure 15: Latency vs achieved throughput on a VM in AWS with increasing random read (top) and sequential read (bottom) load.

6.2 Read Performance in Cloud Deployments

We next deployed a VM in the AWS public cloud containing 128 cores and 512GiB of DRAM, and used iSCSI clients to send a load of 8KiB random reads and 64KiB sequential reads, using a 2:1 compressible dataset. We attached io1 EBS [1] volumes to the VM over the network, exposed as NVMe SSD drives, with an EC2 entitlement of 160K ops/sec. We experimented with ONTAP 9.14.0, in which TopSpin was enabled for cloud VMs. Fig. 15 presents the measurements of server-side latencies and throughput. For random read, an abundance of CPU cores reduces internal queuing times and the benefits of TopSpin, and latency improvements range between 30us and 65us. In contrast, sequential read leverages readahead to hide the drive access times, and TopSpin nearly halves latencies at all loads. TopSpin-enabled cloud deployments will be available using Amazon FSx for NetApp ONTAP (FSxN [2]) later in 2024.

6.3 Mixed Read and Write Workloads

To measure the impacts of TopSpin on mixed read-write workloads and mixed random-sequential workloads, we ran the internal SPC-1 macrobenchmark on the 64-core controller, with clients connected to ONTAP using FCP. SPC-1 issues 40% reads and 60% writes, of which each are 40% sequential and 60% random [17]. These results are shown in Fig. 16, where *Write Baseline* and *Write TopSpin* are the observed write latency without and with TopSpin read enabled, respectively. In this case, overall throughput is not changed through the use of TopSpin, but the same peak throughput is achieved with 4.9% lower CPU usage. Read latency at peak throughput dropped 67%, from 442us to 147us. Further, with

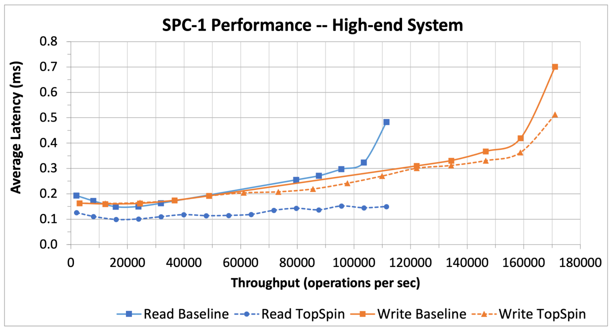


Figure 16: Read and write latencies vs achieved throughput on the 64-core system with increasing SPC-1 load.

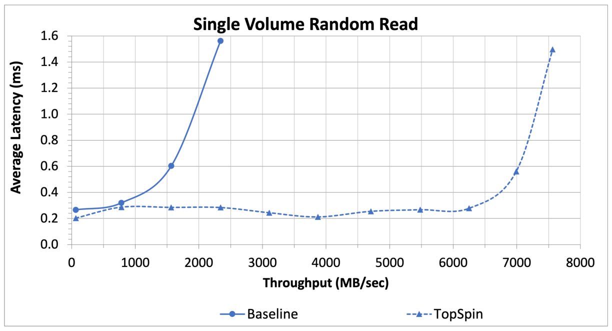


Figure 17: Latency vs achieved throughput on the 64-core system with increasing 8KiB random read load to a single volume.

the majority of reads now bypassing the WAFL component, write latency dropped from 637us down to 517us because wait time for WAFL write messages dropped from 350us to 189us. In this test, 82.4% of reads hit in the cache, 9.2% successfully read from storage, and 8.5% failed through to WAFL due to missing L1s or dirty L0s.

We next evaluated load to an Oracle database. Clients connected over FCP to a 2x18-core controller with 512GiB of DRAM. We compared ONTAP 9.2 to ONTAP 9.3, the first release with TopSpin. Load was generated to an Oracle 12c database using SLOB2 [9], comprising 75% SELECT and 25% UPDATE SQL commands. Peak throughput from the storage server increased from 345K I/Os per second to 400K I/Os per second. As explained earlier, WAFL is designed to complete writes quickly, so UPDATEs do not impact user sessions much. However, storage read latencies directly impact SELECTs, which dropped from 1.13ms to 0.95ms as reported by the database server.

6.4 Additional Benefits to Bypassing WAFL

Beyond the benefits already discussed, TopSpin also provides an effective way to work around a long-standing bottleneck in WAFL parallelism. The Waffinity model translates the WAFL file system into a static hierarchy of data partitions [12], with a single active thread per partition. However, the fixed number of partitions at each level of the hierarchy cannot guarantee optimal performance across all workloads. Although rarely encountered, Waffinity-unfriendly workloads are limited by the data partitioning to using a subset of the available cores. Some examples: the entire system load is to a single volume, all load is to a single LUN or file, or sequential read streams to a single LUN or file where consecutive I/Os move lockstep one partition at a time. Dynamically changing the number of data partitions based on observing the current workload is feasible, but has significant technical challenges. By avoiding the WAFL message, TopSpin parallelism for such a workload is limited only by the number of Protocol component threads. To evaluate one such case, we issued an 8KiB random read load from NVMe-oF clients directed to a single volume on the 2x32-core controller. Fig. 17 shows the results. Waffinity has only 9 client-facing data partitions per volume for WAFL read messages, so the baseline system saturates early once WAFL has utilized 9 cores. With TopSpin enabled, ONTAP activates more Protocol threads to use up to 31.5 cores for processing read operations. Combined with lower processing costs and fewer queuing delays, the increased parallelism yields 226.7% higher peak throughput, and even higher gains under 0.4ms. While this is an extreme case, TopSpin improves many similar scenarios with limited WAFL parallelism.

This section provides further evidence of the value of bypassing layers in ONTAP. It also encourages the continued adoption of TopSpin in other code paths. TopSpin has allowed us to incrementally achieve device-proportional overheads without requiring clean-sheet designs.

7 Lessons Learned

Lesson 1: *Bypassing layers for the error-free data path is an effective and safe way to eliminate software overhead in a modular system.* This approach retains useful component divisions and fails through to the component itself for special cases (only the error-free path needs to be optimized). Fail through correctness requires that such cases disregard any changes to message and system state caused by the partial Fastpath execution. Each successful optimization fueled the next project, and continues to do so. The optimization of other file system operations and protocols using TopSpin are in various stages of development and the design presented for reads has provided a strong foundation for these.

Lesson 2: *Incremental optimization for SSD was the right approach for ONTAP*. Before the Fastpath work, it was widely assumed that ONTAP would not be able to achieve device-proportional software overhead. NetApp thus developed and productized the alpha version of a clean-sheet design SSD-optimized storage system called FlashRay. FlashRay was discontinued for two primary reasons: (a) the roadmap to achieve feature-parity with ONTAP was multiple years and (b) the success of the Fastpaths demonstrated that

ONTAP could be optimized to achieve SSD-proportional latencies. Our incremental approach enabled NetApp to productize all-SSD ONTAP systems that were competitive on price and performance, while preserving legacy features. Critically, the WAFL file system architecture was already well-suited for SSD properties. In our experience, building a fast I/O path was significantly easier than building an entirely new file system with a rich feature set.

These lessons are applicable to other legacy systems and can influence designs of storage systems for new media. As new and faster media become available, future systems will need to go further in lowering software overhead. The remaining non-error message hops will need to be eliminated, such as special cases in TopSpin and even for device access. Subsequently, all I/O code paths will need to be further analyzed (such as for cache line misses) and optimized.

8 Related Work

I/O path optimization for low-latency SSD drives is an area of substantial study, notably bringing software overhead in proportion with device latencies. Shin, et al. [57] eliminate interrupt bottom halves and queue running contexts in the I/O completion path. BarrierFS [65] reduces software overhead by replacing expensive storage device I/O order guarantee approaches. ReFlex [35] builds a highly-optimized, run-to-completion execution model for remote NVMe Flash storage on top of the IX dataplane OS [4]. With only 21us software overhead, ReFlex is fast.

Kernel bypass is another popular approach. NVMeDirect [33] allows user-space applications direct access to the I/O device. Demikernel [69] is a datapath OS that uses kernel bypass devices and an optimized core scheduler for microsecond-scale latencies, even while retaining critical OS functionality. XRP [71] allows the user to embed application logic within the device driver's interrupt handler using eBPF. These hooks include file system state that can traverse on-disk structures and initiate new I/Os without returning control back to the application. These approaches are largely orthogonal to our work because the components of ONTAP discussed in this paper all run inside the kernel.

Techniques to reduce software overheads for low-latency I/O devices include RAID optimizations [63], CPU-scalable drive access [45], transparent zero-copy [59], and overlapping processing with device access [40]. i10 [21] provides a CPU-efficient RDMA remote storage stack, which minimizes the number of cores required to saturate both network and storage devices. SpanFS [24] partitions the file system into independent micro-services by file and directory to increase parallelism of the storage software, which is somewhat similar to Waffinity. Blk-switch [22] treats the storage stack like a network switch, and adapts networking optimizations to minimize software overhead and maximize drive throughput. Many systems optimize for predictable latency from SSD drives [5, 20, 62, 70, 34, 31, 61, 43, 56, 58, 22, 35], some using machine learning [18]. Fast core scheduling can provide QoS at microsecond granularity to latency-sensitive applications [49, 16].

Previous work analyzed low-latency drive performance [36] and its impact on the Linux storage stack [53]. Oh, et al. [48] optimize Ceph to adapt from HDDs to SSDs. I/O schedulers have been optimized [68] for low-latency devices and even evaluated as software overhead [64]. Performance requirements of Key-Value stores have inspired optimizations for these drives, including optimized CPU usage [41, 37, 42] and CPU bypass [46]. Lastly, persistent memory technologies place even more emphasis on low processing costs [6, 66, 39, 23], kernel bypass [8, 7], and indexing overheads [39, 23, 47, 44].

Our work was done on a 30+ year old legacy system without compromising the dozens of enterprise quality features that are critical to our customers. The interactions of our improvements with these features created additional challenges in our designs and implementations. We achieved significant performance gains while retaining the existing behavior of millions of lines of WAFL and ONTAP code outside the read path, despite potentially operating on the same data.

9 Conclusion

Although several aspects of the decades-old ONTAP architecture were well-suited for building an all-SSD controller, the software delays (proportional to HDD latencies) in its legacy I/O path had made that impractical. In this paper, we presented the multi-year journey of incremental improvements to the read path that have reigned in software overhead and made the all-SSD ONTAP controller a success. In future work, we plan to present data reduction technologies that were also crucial to this productization, as well as extensions of TopSpin to other operations, such as writes.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Angelos Bilas, whose suggestions have significantly improved this paper. This paper discusses the results of a decade of engineering effort by a huge group of talented engineers: Abdul Basit, Joseph Brown, Yong Cho, Roopesh Chuggani, Peter Denz, Ravi Dronamraju, Manish Katiyar, Rajesh Khandelwal, Aditya Kulkarni, Szu-wen Kuo, Asif Pathan, James Pitcairn-Hill, Parag Sarfare, Girish Hebbale Venkata Subbaiah, Ananthan Subramanian, Venkateswarlu Tella, Daniel Ting, Sriram Venketaraman, Jungsook Yang, Xiaoyan Yang, and many others.

References

[1] Amazon. Amazon elastic block store. `https://aws.amazon.com/ebs/`.

[2] Amazon. What is Amazon FSx for NetApp ONTAP? `https://docs.aws.amazon.com/fsx/latest/ONTAPGuide/what-is-fsx-ontap.html`.

[3] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing*, 1(1), 2004.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, October 2014.

[5] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[6] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[7] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, February 2021.

[8] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for In-Memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, July 2017.

[9] Kevin Closson. SLOB resources. `https://kevinclosson.net/slob/`.

[10] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal parity for double disk failure correction. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, March 2004.

[11] Storage Performance Council. Storage performance council-1 benchmark. `www.storageperformance.org`.

[12] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] Matthew Curtis-Maury, Ram Kesavan, and Mrinal Bhattacharjee. Scalable write allocation in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2017.

[14] Peter Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2016.

[15] John K Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A Smith, et al. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[17] Binny Gill. SPC-1 benchmark. `https://www.usenix.org/legacy/events/fast05/tech/full_papers/gill/gill_html/node28.html`.

[18] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter Technical Conference*, 1994.

[20] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, February 2017.

[21] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP=RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, February 2020.

[22] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '19, 2019.

[24] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, July 2015.

[25] Ram Kesavan, Matthew Curtis-Maury, and Mrinal Bhattacharjee. Efficient search for free blocks in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2018.

[26] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Storage gardening: Using a virtualization layer for efficient defragmentation in the wafl file system. In *17th Usenix Conference on File and Storage Technologies (FAST)*, 2019.

[27] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Countering fragmentation in an enterprise storage system. *ACM Transactions on Storage*, 15(4), jan 2020.

[28] Ram Kesavan, Harendra Kumar, and Sushrut Bhowmick. Wafl iron: Repairing live enterprise file systems. In *16th Usenix Conference on File and Storage Technologies (FAST)*, 2018.

[29] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[30] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (ToS)*, 13, October 2017.

[31] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an autonomic SSD architecture. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[32] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing array of SSDs when the storage device is no longer the performance bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, July 2017.

[33] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, June 2016.

[34] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O path: A holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, February 2017.

[35] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash = local flash. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 18)*, ASPLOS '17, April 2017.

[36] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of Ultra-Low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, July 2018.

[37] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, February 2019.

[38] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th Usenix Conference on File and Storage Technologies (FAST)*, 2017.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '17, 2017.

[40] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019.

[41] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '19, 2019.

[42] Haoyu Li, Sheng Jiang, Chen Chen, Ashwini Raina, Xingyu Zhu, Changxu Luo, and Asaf Cidon. RubbleDB: CPU-Efficient replication with NVMe-oF. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023.

[43] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. Ioda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '21, 2021.

[44] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, February 2022.

[45] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, July 2021.

[46] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, June 2013.

[47] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, February 2021.

[48] Myoungwon Oh, Jugwan Eom, Jungyeon Yoon, Jae Yeun Yun, Seungmin Kim, and Heon Y. Yeom. Performance optimization for all flash scale-out storage. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[49] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, February 2019.

[50] Naresh M. Patel. Half-latency rule for finding the knee of the latency curve. *SIGMETRICS Perform. Eval. Rev.*, 43(2), sep 2015.

[51] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002.

[52] Samsung. Pm1725 nvme pcie ssd. `https://www.samsung.com/us/labs/pdfs/collateral/pm1725-ProdOverview-2015.pdf`.

[53] Eric Seppanen, Matthew T. O'Keefe, and David J. Lilja. High performance solid state storage under linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2010.

[54] Skip Shapiro. Technical report: Flash cache best practice guide. `https://www.netapp.com/pdf.html?item=/media/19754-tr-3832.pdf`.

[55] Skip Shapiro. Technical report: Flash pool design and implementation guide. `https://www.netapp.com/pdf.html?item=/media/19681-tr-4070.pdf`.

[56] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, June 2013.

[57] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, June 2014.

[58] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: Consistent flash performance through redundancy. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, June 2014.

[59] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, July 2022.

[60] Rajesh Sundaram. The Private Lives of Disk Drives. https://www.netapp.com/atg/publications/publications-the-private-lives-of-disk-drives-20064017/, 2006.

[61] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, July 2021.

[62] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika MansouriGhiasi, Lois Orosa, Juan Gomez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[63] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. StRAID: Stripe-threaded architecture for parity-based RAIDs with ultra-fast SSDs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, July 2022.

[64] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. Do we still need io schedulers for low-latency disks? In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 23)*, HotStorage '23, 2023.

[65] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, February 2018.

[66] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2016.

[67] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *INFLOW*, 2014.

[68] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, SOSP '15, 2015.

[69] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195?211, New York, NY, USA, 2021. Association for Computing Machinery.

[70] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching through server storage stack from kernel to firmware for Ultra-Low latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, October 2018.

[71] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, July 2022.