FlashNet: Flash/Network Stack Co-Design

Animesh Trivedi[1], Nikolas Ioannou[1], Bernard Metzler[1], Patrick Stuedi[1],
Jonas Pfefferle[1], Ioannis Koltsidas[1], Kornilios Kourtis[1], and Thomas R. Gross[2]
[1]IBM Research and [2]ETH
Zurich, Switzerland

ABSTRACT

During the past decade, network and storage devices have undergone rapid performance improvements, delivering ultra-low latency and several Gbps of bandwidth. Nevertheless, current network and storage stacks fail to deliver this hardware performance to the applications, often due to the loss of IO efficiency from stalled CPU performance. While many efforts attempt to address this issue solely on either the network or the storage stack, achieving high-performance for networked-storage applications requires a holistic approach that considers both.

In this paper, we present FlashNet, a software IO stack that unifies high-performance network properties with flash storage access and management. FlashNet builds on RDMA principles and abstractions to provide a direct, asynchronous, end-to-end data path between a client and remote flash storage. The key insight behind FlashNet is to co-design the stack's components (an RDMA controller, a flash controller, and a file system) to enable cross-stack optimizations and maximize IO efficiency. In micro-benchmarks, FlashNet improves 4kB network IOPS by 38.6% to 1.22M, decreases access latency by 43.5% to 50.4 μsecs, and prolongs the flash lifetime by 1.6-5.9× for writes. We illustrate the capabilities of FlashNet by building a Key-Value store, and porting a distributed data store that uses RDMA on it. The use of FlashNet's RDMA API improves the performance of KV store by 2×, and requires minimum changes for the ported data store to access remote flash devices.

CCS Concepts

•Information systems → Storage network architectures; Flash memory; •Networks → Network performance evaluation; •Software and its engineering → Operating systems;

Keywords

RDMA; Netwoked Flash; Performance; Operating Systems

SYSTOR'17, May 22-24, 2017, Haifa, Israel

ISBN 978-1-4503-5035-8/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3078468.3078477

1. INTRODUCTION

Modern distributed applications such as data processing stacks (Spark or MapReduce), web services, databases, etc. routinely store, access, and analyze Terabytes of data stored across hundreds of storage servers. Consequently, their performance depends considerably on the IO performance of the many storage *and* network devices involved. Thankfully, the IO performance of network and storage devices has been dramatically improved over the last decade. Ethernet has evolved from 1 to 10, 40, and now 100 Gbps data rates with single-digit microsecond link latencies. Flash storage has delivered 2-4 orders of magnitude bandwidth and latency improvements over HDDs, while recent advancements in non-volatile memory technologies [55] promise to improve performance further by one or more orders of magnitude.

However, translating these raw IO performance improvements into application-level gains remains a challenge due to multiple factors. First, traditional IO stacks[1] are designed assuming *slow* IO and *fast* CPUs, which is no longer true [55]. Consequently, the cost of maintaining generic OS abstractions, IO interfaces, scheduling, context switches, implementation-related inefficiencies, etc., contributes significantly to the loss of IO efficiency. Second, recent improvement efforts either target the network [29, 49, 22] or the storage stack [66, 83, 4, 35] exclusively, but not the combination of both. As a result, remote data accesses over the network typically involve costly application-level coordination across network, storage, and file system operations. Last, many established solutions in this space, such as NFS or iSCSI, aim to deliver data only up to the file (NAS) or block (SAN) level. Hence, they still leave last-mile, end-host inefficiencies between data and clients, as we demonstrate in the next section. In conclusion, there is a need for a *holistic* approach that tackles both network and storage challenges to enable high-performance remote data accesses (not just file, block, or device) between servers and their clients.

In this paper, we present FlashNet, a co-designed IO stack that delivers high-performance data access to networked clients. The FlashNet stack builds upon the data and control path separation philosophy of Remote Direct Memory Access or RDMA networks. RDMA networks have already shown to deliver high network performance to various applications [69, 33, 57, 52, 14], and related concepts [7, 82] and even APIs [74] are explored in the storage domain as well. In FlashNet, we unify these fragmented efforts across the IO stacks, and extend the path separation philosophy of RDMA networks to storage with the help of a co-designed flash file

[1]collectively referring to the network and storage stacks.

system and a flash device controller. This extension enables FlashNet to establish an *end-to-end* network data path between clients and data on remote flash devices, thus eliminating any last-mile, end-host inefficiencies, which is the hallmark of RDMA networking. Furthermore, a co-designed storage/network stack also means that FlashNet can target optimizations across the stacks to reduce overheads. For example, FlashNet issues flash IO directly from the network stack, tracks dirty data between network and storage stacks in a unified manner, uses network access statistics to better manage flash devices, etc. As a result, FlashNet delivers 1.22M IOPS (100% of 40Gbps network bandwidth performance), decreases access latencies by 43.5% to 50.4 μsecs, and improves the flash lifetime by 1.6-5.9× for writes.

To illustrate the benefits of FlashNet's RDMA API, we have built one application, and ported one RDMA-ready application on it. Our first application is a Key-Value (KV) store that uses RDMA operations to access data from a remote flash storage in a disaggregated setting. We chose this workload because previous work in this space has identified network and last-mile inefficiencies with heavy-weight protocols such as iSCSI to be a bottleneck [36]. By using FlashNet's one-sided RDMA read/write operations, the KV store delivers 490K IOPS, representing a 102.4% improvement from the baseline number of 242.7K IOPS achieved over sockets and files. Our second application is a distributed in-memory data store called RStore [73], which uses RDMA to access data from remote DRAMs. We have ported it and one of its applications, a distributed Sorter, to FlashNet. The porting process requires minimum changes to RStore. In comparison to the original in-memory version, FlashNet imposes no performance overheads to the run-time, and the ported Sorter delivers a performance which is the sum of its in-memory run-time and flash read/write time.

Our specific contributions include (a) proposing and extending the path separation philosophy of RDMA for remote flash storage accesses; (b) FlashNet, a co-designed prototype IO stack that consists of a RDMA network controller, a file system, and a flash controller; (c) evaluating FlashNet in a distributed setting, showcasing that it reduces CPU overheads to deliver peak IO performance, helps in translating raw performance into application-level performance, and adds minimum overheads to RDMA-ready applications.

2. MOTIVATION

We first quantify overheads associated with remote data accesses. We consider a storage server that serves client requests to access data from a storage device over the network (e.g., an iSCSI storage node, or a key-value server). The server's peak performance depends on the efficiency of both the network and the storage operations. Our setup consists (for details see Section 5) of a server with three off-the-shelf PCIe NVMe drives in a software RAID-0 configuration and an `ext4` file system on top. The server is connected to clients over 40 Gbits/sec Ethernet. The clients issue 4kB random reads from remote NVMe devices over the network. We measure the peak IO operations per second (IOPS) delivered by the server for the following configurations:

- *iSCSI:* 3 NVMe devices as iSCSI targets (partitioned between clients) using in-kernel iSCSI drivers, tuned as [36].
- *NFS:* An in-kernel NFS server that accesses data via an `ext4` file system on top of the software RAID-0 device.

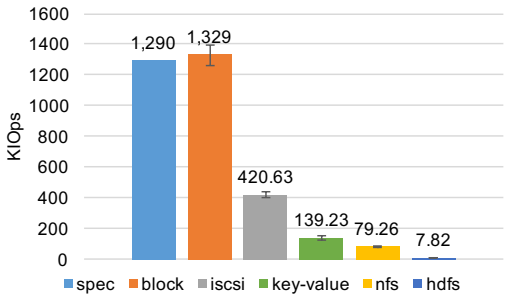

Figure 1: Random 4kB read IOPS performance.

- *Key-Value:* We evaluate the performance of Aerospike KV, a state-of-the-art NoSQL store [68]. Aerospike is configured to run over the raw block device and recommended high-performance configuration (see Section 5.2.1).
- *HDFS:* We measure Hadoop file system (HDFS) clients accessing a random 4kB block in a private 8GB file.

We generate load for file and block IO with `fio` [16]. We include performance numbers from the device specification marked as `spec` and local block-level IO performance. The specification for each device is 430K IOPS, reaching an aggregate of 1.29M IOPS. There are three key points in our results (Figure 1). First, none of the configurations were able to deliver the raw NVMe device performance to clients across the network. iSCSI peaks at 420.6K, NFS at 79.2K, Aerospike at 139.2K, and HDFS at 7.8K IOPS. Second, in most of the configurations, the performance is limited by the CPU performance. In Figure 2 we illustrate various IO paths involved between a server and a client. Along these lines, the CPU has to to do network processing, IO buffer management, file system execution, block IO, scheduling, and execute generic OS services and abstractions. In line with previous findings [36], our analysis of iSCSI reveals that performance is limited by CPU load. For NFS, overheads come from the protocol/buffer management, RPCs, and file system related overheads where the NFS server has to serve files from the `ext4` file system. Application-level workloads, i.e. KV and HDFS, are more involved than a simple block or file-level access, requiring multiple round-trips over the network. Hence, their peak performance is driven by the application logic, which in absence of a direct access remote storage API, requires the server application to be actively scheduled to orchestrate the data flow. The cost of scheduling, context switches, cache flushes, IO resource management (as they are tied to the process abstraction), etc., becomes an overhead. And lastly, though efforts have been made to remedy exactly these overheads, they only focus on either network or storage, but not both. In networking, for example, the use of RDMA has been proposed to reduce end-host overheads [52, 14, 77, 34]. Similarly, on the storage side, multiple projects advocate to eschew the OS in the data access path in favor of a leaner and faster access to flash storage [7, 42]. FlashNet is built on similar principles, but goes a step further taking a holistic approach for building a network- and storage-wide solution for fast data accesses.

3. DESIGN OF FLASHNET

FlashNet is a unified *software* stack that consists of three logical components: a flash controller, a file system (Con-

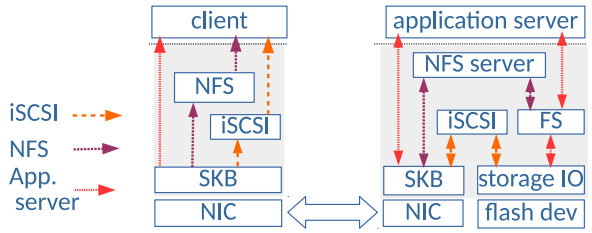

Figure 2: IO paths taken by data in tested settings.

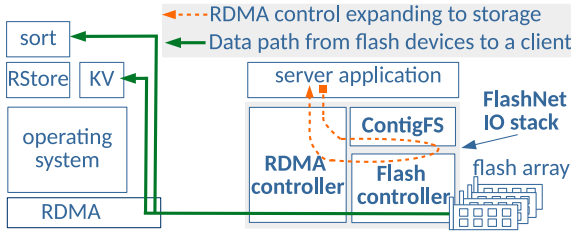

Figure 3: FlashNet stack illustrating the network-storage setup and the end-to-end data flow path.

tigFS), and an RDMA controller. These components are co-designed to eliminate IO inefficiencies in an *end-to-end* manner when data flows between a remote flash device and a client buffer. Figure 3 shows the setup and interaction among these components. Applications use FlashNet by programming against a familiar RDMA API, and FlashNet transparently extends RDMA operations (originally intended for DRAM buffers) to flash devices. Applications already using RDMA require minimum changes to access data from remote flash devices. The design of FlashNet is guided by three principles:

1. *Eliminate application involvement from IO flows:* FlashNet leverages the path separation philosophy of RDMA, extending it to storage devices to completely eliminate an application's involvement on data transfers.
2. *Minimize system overheads:* FlashNet utilizes a file system with a simple file layout so that costly file system operations (e.g., inode look-ups, checks, location translations, etc.) are eliminated from the data path between the network and storage controllers.
3. *Keep application interface simple and clean:* FlashNet leverages traditional files and `mmap`-based IO, to autonomously and efficiently manage data transfers over RDMA networks. An RDMA-ready application does not have to use any new interface or abstraction.

3.1 The Flash Controller

A key part of the FlashNet architecture is its flash controller design. A flash controller manages the performance and packaging idiosyncrasies of flash devices. However, prevalent embedded flash controller designs are too restrictive for the FlashNet architecture due to multiple reasons. First, with high-speed networks, a flash page containing hot data may experience bursts of concurrent small writes from the network within a small time frame (a few μsecs). Even though the networking stack contains pertinent information which could be useful to absorb the bursty, concurrent nature of network IO, there is no standard way to pass this information to the flash controller for better flash management. Second, flash devices are exposed as conventional block devices where the logical block management is tied with the flash storage management. Previous research in the field has demonstrated that decoupling these two can lead to performance improvements with a much simpler file system layout [31]. A simpler file layout enables removing the file system from network data transfers. And lastly, the one-controller-per-device design cannot leverage multiple flash devices present in a system.

To alleviate the aforementioned restrictions and to jointly optimize the flash controller with the rest of the stack, we have designed and implemented a software-defined flash array controller that builds on top of virtualized flash storage works [78, 31]. The controller decouples the logical block management from the flash storage management and exports a large 64-bit block address space using a virtualized Flash Translation Layer (FTL). The FTL dynamically maps logical block addresses (LBAs) to physical block addresses (PBAs) over a virtual device array made out of one or more flash devices. An LBA entry in the FTL contains the location of the data on a flash device (its PBA) and its location in a DRAM buffer (if the data is present in the system). This design ensures that all concurrent accesses (networked or local via the file system) are given the same DRAM page or PBA location. All accesses to data happen through a `get/put` page interface (see Table 1). This interface also allows the flash controller to track heat and access frequency information related to data/page accesses.

Dirty data is always written out-of-place while keeping track of new LBA to PBA mappings in the FTL. Updates to the FTL are appended to the flash device asynchronously with the data and are synced when instructed by an application or a remote RDMA access. The controller uses a log-structured allocation strategy to allocate PBA blocks across multiple devices. It ensures uniform wear leveling, and employs advanced data placement and efficient garbage collection (GC) policies to reduce write-amplification. Our GC algorithm employs a greedy policy [11], using a per-block reference count for validity tracking and a reverse map for blocks that are not fully invalid. Under non-uniform (i.e., skewed) workloads, the controller segregates data into three data streams based on their update frequency to reduce data relocation overheads [26]. Our controller performs a three-level data segregation scheme based on (a) the logical origin of data blocks; (b) their age in the system; (c) their frequency and heat of updates tracked with the `get/put` API.

To protect against crashes, the flash controller logs updates to the mapping table (LBA-to-PBA). Upon initialization, it first checks whether it was cleanly shut down using checksums and unique session identifiers written during the LBA-to-PBA dumps. If a clean shutdown is detected, the mapping table is fully restored from a fixed location. Otherwise, the mapping table is re-constructed by scanning all the log metadata pages, based on back-pointers and timestamps.

3.2 Contiguous File System (ContigFS)

As storage devices get faster, the constant and unnecessary involvement of a file system in every aspect of IO (local or networked) operations generates a significant amount of overhead [7, 40]. In the IO path, one of the key file system operations is the translation of a file offset to a device block location. With the current extent-based file layouts, it is difficult to reduce the file system involvement from the IO path

due to their sophisticated extent management logic. However, by co-designing the file system together with the storage controller (with its virtual 64-bit FTL address space), we can simplify the file system design by using a range-based rather than an extent-based file layout. A range-based file layout stores a complete file in contiguous device block addresses. This layout enables a trivial file offset to device location translation by adding the file offset to the start location of the file on the device. This translation can also be done by the network controller, hence removing the file system and application involvement from the networked IO. As an alternative, raw block IO can be used to remove the file system from the IO path. However, this option eliminates many applications that use hierarchical naming and access control (e.g., HDFS directories) from running on FlashNet.

To realize our idea, we design a POSIX file system called Contiguous file system or ContigFS that does contiguous file allocations on top of the virtualized FTL address space. The files that are stored in a contiguous LBA address range can grow and shrink by manipulating their mappings in the FTL address space. Like any other file system, ContigFS provides the full file system API to applications. To exert full control over data buffering and sharing to/from flash devices, ContigFS co-manages (with the flash controller) its own pool of DRAM pages. Data is staged for access and dirty data is written out from pages in the pool. Pages from the pool are also given to serve page faults in `mmap`'ed memory regions. In a similar spirit to the IO-lite system [58], ContigFS ensures (in collaboration with the flash controller) that there is only a single physical and consistent copy of data in the system that is shared between the storage controller, the network controller, and applications.

3.3 The RDMA Controller

The RDMA controller of FlashNet extends the data and control paths [80, 76] of RDMA networks to include the file system and the flash controller as well. Similarly to the original path separation idea, applications, file systems, and, to a large extent, the OS, are eliminated from the extended data flow path. In order to achieve this, FlashNet translates necessary abstractions in advance from a RDMA access to the data location on a flash device. This means that an RDMA access to a memory address can be mapped to its flash LBA directly by the controller without having to consult the application or the file system.

A key operation on the extended control path is the RDMA buffer registration process. In the buffer registration process, every data source or sink buffer is preregistered with the RDMA stack to generate a buffer identifier called *Steering Tag* or STag. This STag is used in subsequent RDMA operations to identify network source or sink buffers without involving the application to steer data flows. FlashNet uses the same mechanism to identify files and offsets to resolve data locations which are involved in a network operation. ContigFS files, which are involved in RDMA network operations, are registered with the FlashNet RDMA controller by passing their `mmap`'ed area. At this point, with the help from the ContigFS, the RDMA controller translates the memory area to the start LBA of the file. As files are contiguously allocated in the LBA address space, further offset calculations during network operations are done entirely by the RDMA controller and then passed to the flash controller for reading/writing data from/to involved LBAs. Hence, on the fast data path, the file system is eliminated and the two device controllers talk to each other to manage data flows. Figure 4 shows the end-to-end translation process between these abstractions on the extended control and the data path.

Figure 4: Abstraction translations inside FlashNet.

We cannot leverage existing RDMA NICs (RNICs) because they require pages to be immediately pinned. This setup (a) does not allow the system to track dirty data from RDMA access (as they happen in hardware and are transparent to the system) and ensure it is flushed properly; (b) does not allow scaling beyond the system DRAM size; (c) is wasteful to the system DRAM. To overcome these limitations, the RDMA controller of FlashNet is implemented in software and supports on-demand memory pinning. With on-demand memory pinning, during the memory registration (happens only for memory segments backed from a ContigFS file), the controller only allocates necessary metadata, locks, and data structures to hold DRAM page pointers, but does not pin pages yet. The pages are populated and pinned on-demand on the extended data path when an RDMA request accesses them. Types of access to these pages (read or write) are told to the flash controller to track dirty data. These pages are also shared concurrently (without creating copies) between an application, the network, and the storage stack using RDMA buffer ownership rules.

3.4 The Life of an IO Operation

In this section, we illustrate an example where a server serves data to a networked client using FlashNet's one-sided RDMA read operation in Figure 5. The server application first starts by creating files on ContigFS (not shown) using `fallocate`. While processing the file allocation request, ContigFS requests a contiguous LBA address range from the flash controller to store data. Upon a successful file creation, the server process then `mmap`s these files into its address space (step ❶). It then registers the `mmap` address with the FlashNet RDMA controller to prepare it for RDMA operations (step ❷). The controller resolves the passed region to be a ContigFS-file region and hence, only translates mappings and saves the LBA address of the memory region by adding the `mmap` offset to the starting LBA address of the file (step ❸). The controller then generates a valid STag. The server then communicates the relevant RDMA credentials that include permissions, `mmap` addresses, and STags to a client. Steps ❶–❸ constitute the extended control path of the FlashNet stack where the file system, the flash controller, and the RDMA controller work in unison to establish mappings and translate abstractions.

On the extended data path (steps ❹–❾), a client, after having acquired right RDMA credentials from the server process, issues an RDMA read request. Upon receiving an incoming RDMA read request, the RDMA controller first resolves the flash buffer using the STag present in the request. The controller then calculates the flash LBA address by adding the offset (present in the RDMA request) to the previously saved base LBA address of the registered region

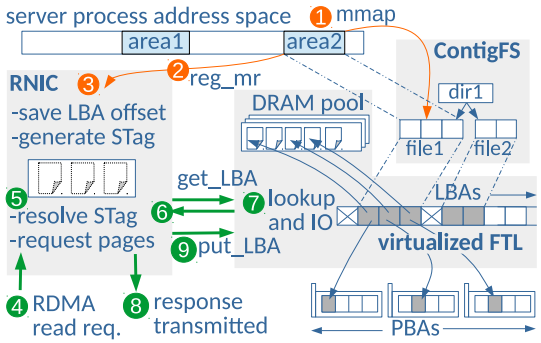

Figure 5: The life of a FlashNet RDMA read.

(step ❺). The flash pages in the identified region are then *populated* (see Section 4.2) with the help of the flash controller (steps ❻ and ❼). Upon completion of the RDMA request processing (step ❽), the involved LBA pages are given back to the flash controller (step ❾). Meanwhile, the incoming RDMA response data is directly deposited in the client's buffer by the RDMA controller. The server-side data processing does not involve any active involvement or coordination from the server application.

4. IMPLEMENTATION OF FLASHNET

The components of the FlashNet stack are implemented in software as Linux kernel modules and do not rely on any specific hardware support. The implementation of the virtualized flash controller is based upon the SALSA software flash controller [28], which is extended to support the in-place, DRAM-based page sharing, and callback based non-blocking IO API. ContigFS hooks into the VFS layer of the kernel and talks to the flash controller for the block management. The RDMA controller of FlashNet is derived from the open-sourced SoftiWARP RDMA controller [50, 72], which provides complete iWARP RDMA features and semantics. It uses memory-mapped IO queues and non-blocking kernel TCP sockets with per-core kernel threads to perform asynchronous network IO on behalf of user processes. SoftiWARP is enhanced to interact with ContigFS and the flash controller to support lazy memory management while maintaining the compatibility with current RDMA RNICs for mixed RNIC-FlashNet deployments. The whole FlashNet framework *does not* require any changes to the kernel.

4.1 File Management

ContigFS file management is similar to the Direct File System [31, 71]. One key difference being that with the current prototype we do not reserve LBA ranges to provide large unassigned LBA ranges to files and let them grow in that. Instead, ContigFS performs remapping on the FTL as the file size grows out of its current allocation size. The current size is set by a `fallocate` call, which commits LBA space for the given file size. The size of an LBA block is configurable (default is 4kB). The file can grow and shrink by `reallocing` its LBA address in the FTL. If necessary, files can be relocated in the FTL address space without physically moving the data on the devices by updating the FTL mappings of the new LBA range to the old PBA entries. The LBA space is managed using the buddy allocation technique [38].

ContigFS Layout: ContigFS splits the 64-bit virtual FTL address space into two regions, saving file metadata and data separately. The metadata region provides a mapping from inode numbers to 32 bytes of metadata that include the data LBA address (on the second region), the file size, file permissions, and flags. For directories, the data LBA address contains the directory entries. The metadata region is implemented as a file, and can be expanded as needed. Metadata changes are made persistent synchronously with respect to file modifications. The ContigFS design is optimized for the FlashNet workloads, targeting large files and shallow directory hierarchies.

API functions	Description
is_contigfs_vma(va, len)	is `vma` a ContigFS mmap'ed area
get_LBA(addr, len, flags, cb*)	gets DRAM pool pages for IO
put_LBA(addr, len, flags, cb*)	puts pool pages in a LBA range

Table 1: The flash controller (abridged) API.

4.2 Flash Page and Buffer Management

Data from flash devices is buffered and shared on DRAM pages from the pool, which is co-managed by ContigFS and the flash controller. These pages are populated using a set of asynchronous `get` and `put` based interfaces (Table 1) to the flash controller. The `get_LBA` call takes a start LBA address, size, and type of request (r/w) with flags, and provides DRAM pages from where data can be read from or written to. Obviously, a request for a write access marks the page dirty. After usage, these pages are put back to the flash controller using the `put_LBA` call. Both calls are byte-addressable and take a callback pointer (marked as `cb`) to indicate the completion of an operation.

Flash Page State Machine: To ensure that no two entities in the system see different data, the flash controller implements a state machine with *atomic* transitions (shown in Figure 6) for every flash LBA page. The state of an LBA page is stored with its FTL mapping. An uninitialized flash LBA page starts in an `Invalid` state. At the time of the first write, the controller allocates a PBA address, maps it to the LBA page, and updates the LBA entry state in the FTL from an `Invalid` to a `PBA` state. For a get request, the flash controller checks the state of the LBA pages involved to determine if any page has previously been brought into the system (i.e., contains an LBA entry in the FTL that points to a DRAM page in the buffer pool or a device PBA). For LBA pages that are not in the buffer pool (indicated by the `PBA` state), the flash controller allocates new DRAM pages from the pool, issues DMA requests for them, and atomically updates their status to `IN_FLIGHT`. When the DMA is finished (`IO_DONE`), the associated DRAM page is atomically installed in the FTL and a callback is executed with the DRAM page address. Any subsequent read or write get requests on this LBA is given the same DRAM location while maintaining usage and frequency counters on a per-page basis. These counters are used by the flash garbage collector to identify the hot and cold LBA pages. For LBAs which were already in the buffer pool, the flash controller immediately issues callbacks to the requesting entity (either the RDMA controller or ContigFS) with a valid DRAM page pointer. Consequently, depending upon the status of the pages involved in a request, callbacks can be issued in any order. After processing, references to pages are put down by calling `put_LBA`. Additionally, for dirty pages, a call to `put_LBA` can optionally generate a callback when the LBA

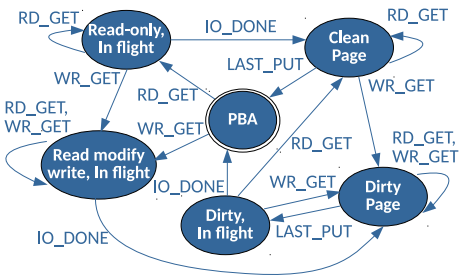

Figure 6: The state machine of a flash LBA page.

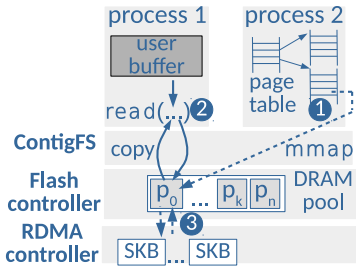

Figure 7: Semantics of a page sharing between (1) `mmap`; (2) POSIX IO; (3) remote RDMA accesses.

was made persistent (i.e., transitioned to a `PBA` state). Concurrent small writes are absorbed by the same DRAM page, and only the last put call triggers a dirty data write out. The current flash controller prototype does not cache data. A selected form of time-bounded caching is done for pre-fetching. The pre-fetching logic is similar to the `get_LBA`, but without the callbacks.

4.3 Data Access and Concurrency

Local data accesses happen using `mmap` or POSIX `read` or `write` calls on ContigFS. When a process calls an `mmap` (path 1 in the figure), the Linux kernel allocates a contiguous virtual memory area (VMA) within the process address space and passes it to ContigFS. Upon receiving the call, ContigFS does sanity and permission checks and registers itself as the page fault handler for this area. When the process accesses the data and a page fault happens, ContigFS is notified and it calculates the LBA from the faulting address. It then calls `get_LBA` with the LBA, and installs the DRAM page from the pool provided in the callback into the application address space. For `read` or `write` calls (path 2 in the figure), the same LBA calculation step from a file offset is followed and data is copied into the process provided user buffers. Pages are put back to the pool after copying or when the process calls `munmap`.

Remote networked clients access data by issuing RDMA operations. As outlined in Section 3.3, the RDMA controller also calculates the target LBA address by the simple offset calculation based on the address present in the RDMA request (path 3 in the figure). The RDMA controller uses this LBA address to issue asynchronous `get_LBA`. If the target LBA address is not present in the pool, the RDMA network processing can be stalled. A stalled connection is resumed after receiving callbacks form the flash controller. However, as explained in the previous section, these callbacks can happen in any order whereas the RDMA specification requires in-order data processing. In order to keep track of out-of-order page callbacks, FlashNet takes advantage of the fact that RDMA requires pre-allocation and registration of flash VMAs. At the time of registration, FlashNet allocates an atomic counter (separately from the flash controller) on a per-page basis to store the validity of the page. In a callback, this counter is increased and before calling a put on the page, the counter is decreased. FlashNet checks the *readiness* of a region by scanning for the longest sequence of non-zero atomic counters and only processes data in that ready region. After the processing, the pages are put back.

The flash controller is the serialization point that provides the LBA's mapping to local and remote accesses. It utilizes the compare-and-swap primitive to implement atomic LBA mapping state transitions (Section 4.2) on aligned 64-bit FTL mapping entries. Thus, concurrent accesses to an LBA will observe coherent values of its mapping (PBA, Dirty, etc.). However, no consistency is guaranteed for the data itself other than what the RDMA semantics dictate.

4.4 Dirty Data Write-Outs

One issue with the use of RDMA one-sided operations to write data is when/how to instruct the storage stack to make data persistent. Recall that there is no application involvement while servicing these one-sided RDMA write requests, and hence it cannot be notified to trigger data write-outs. For this purpose, FlashNet defines *Semantics STags*, which are like normal STag, but they carry additional operational semantics with them. These semantics are masked in lower 8-bits (reserved for application use by the RDMA specification) of a 32-bits STag to set `sync` or `async` flags provided by FlashNet. With this extension, upon encountering a `sync` flag in an STag, the RDMA controller does not process further incoming RDMA operations on a particular connection until dirty data from the last operation is written to the flash device (indicated by the `put_LBA` callback). Consequently, no work completion notifications are generated on the client side. Whereas `async` write processing continues immediately without waiting for the put callbacks, and a client-side work completion is generated immediately. Independently of the way incoming RDMA writes are processed, the client-side RDMA queues always remain asynchronous. Applications built on top of FlashNet requiring strong durability guarantees can utilize the `sync` *Stag*; it provides similar guarantees to the REQ_FLUSH block request flag in Linux.

5. EVALUATION

We evaluate FlashNet on a cluster of 17 machines with dual Xeon E5-2690 CPUs, 256 GB of DRAM, and Chelsio T5 40 Gbps NICs, running Linux 3.19 on Ubuntu 15.04. One machine has three off-the-shelf, enterprise-level NVMe PCIe flash devices on which we run FlashNet. The remaining machines run clients that repeatedly request the server for data stored in flash. Clients use SoftiWARP as a software RDMA device [50]. All numbers reported here are the average of of 3 runs, each lasting 60 seconds. FlashNet experiments are done with the files on ContigFS while other configurations use the `ext4`. The key performance highlights are:

- FlashNet successfully delivers high performance to networked clients for remote NVMe accesses. It scales well with the number of clients, and with 256 clients saturates the 40 Gbps network with 1.22M 4kB IOPS.

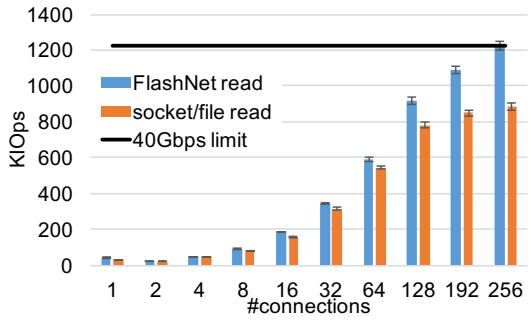

Figure 8: 4kB random read performance.

- The direct end-to-end data path of FlashNet helps to reduce the network read latency of a 4kB flash page from 89.2 μsecs to 54.9 μsecs, a 38.4% improvement. This latency can be reduced further by a margin of 8.2% to 50.4 μsecs in a mixed RNIC-FlashNet setup.
- By efficiently managing the flash device using the access, heat, and frequency data provided by the flash controller's API, FlashNet reduces the write amplification by 38–83% under skewed write workloads.
- The use of FlashNet's RDMA API helps applications to deliver performance close to the hardware. The KV store delivers 460K get IOPS, a close to 2× improvement over its socket/file-based variant. For ported distributed sorting application of RStore, FlashNet adds negligible overheads to access data from remote flash devices.

5.1 Micro-Benchmarks

We first evaluate the performance of FlashNet using micro-benchmarks. We use `netperf` [53] which we augment in two ways: (1) support for RDMA operations; (2) support for accessing flash via POSIX `read/write` calls. Our objective is to eliminate protocol or implementation-related concerns from SAN/NAS solutions, and focus solely on IO operations while highlighting the potential raw performances.

5.1.1 Peak IOPS Scaling

We first revisit our key experiment from Section 2. Figure 8 shows the performance of FlashNet/RDMA operations in comparison to socket/file-based operations in `netperf`. On the X-axis we have a number of clients and the Y-axis shows the number of 4kB random read IOPS delivered. The horizontal bar shows the network performance limit. For a single client, in comparison to a socket/file-based client performance of 33.6K IOPS, FlashNet delivers 42.2K IOPS as peak IOPS. However, for 2 clients, the performance drops due to uneven balancing with 3 NVMe devices and 2 CPU nodes, where one devices gets twice the amount of requests as the other two do. The performance is recovered again after that, and, as evident, both approaches scale well with the number of clients. The socket/file-based approach stops scaling linearly around 64 clients, and hits the peak at 256 clients and delivers 880.9K IOPS. At this point, the server CPU is fully saturated. In comparison, FlashNet is able to deliver 1.22M IOPS, an improvement of 38.6% over a socket/file-based client, and 190% better than the peak iSCSI performance of 420.6K IOPS. The FlashNet performance is only limited by the peak network performance which is saturated at 40 Gbps when delivering 1.22M 4kB IOPS. Hence, FlashNet reaches its first objective of delivering full storage performance across the network.

	net.	sto.	IO	sched.	kernel	proc.	misc.
socket/file	19.3%	7.3%	6.7%	15.8%	40.1%	4.7%	6.1%
FlashNet	20.6%	0.8%	6.4%	8.4%	46.7%	11.7%	5.4%

Table 2: Breakdown of CPU cycles.

Table 2 breaks down the CPU cycles into network (net.), storage (sto.), device drivers (IO), scheduling (sched.), kernel, request processing (proc.), and miscellaneous (misc.). We focus on the cycles spent in storage, scheduling, and request processing. The storage column includes routines from the ext4 file system, the generic VFS layer, and the block layer, etc. Most of these routines are executed for *every* network-storage request. Furthermore, the traditional IO stack architecture experiences high context switching and scheduling related overheads. This is because most of the IO resources (e.g. network sockets, buffers, etc.) are tied to the process abstraction. These resources need to be valid during the IO processing and hence require the application process to be scheduled for IO processing and data movement orchestrations. As a result, not many CPU cycles are left for actual request processing. With FlashNet's design, the file system and most of the generic storage stack is eliminated from the core IO path and request processing does not require server process scheduling. The CPU cycle gains from here are then used in the request processing.

5.1.2 Latency and Bandwidth

One key benefit of RDMA networking is low latency network operations. A FlashNet/`netperf` client is able to read and write a single 4kB random block to a remote flash device in 59.8 μsecs and 83.3 μsecs, respectively. In comparison, the socket/file-based client takes 89.2 μsecs and 114.9 μsecs, respectively. The read performance numbers can be further improved to 54.9 μsecs when we configure the FlashNet device in the latency mode with polling. In the polling mode, the server CPU polls for a few more micro-seconds in anticipation of the next request before going to sleep. We configure this time to be around network RTT.

FlashNet also delivers good bandwidth to its clients. For the bandwidth test, clients read and write 1MB data blocks to a large file. A single socket/file-based client observes 13.6 and 17.4 Gbps of read and write bandwidth, respectively. In comparison, FlashNet delivers 24.4 Gbps (gains: 79.4%) and 25.6 Gbps (gains: 47.1%) of read and write bandwidth, respectively. As we increase the number of clients, both approaches hit the networking bandwidth limits and are able to deliver close to 40 Gbps bandwidths to their clients.

5.1.3 The Cost of Flash Buffer Registration

We now evaluate the cost of buffer registration for FlashNet, which is one of the costliest operation in RDMA [18]. The key difference between standard RDMA and FlashNet registration is the lack of page allocation and pinning costs from the latter. However, per-page metadata are still allocated. Figure 9 shows the cost of `mmap`ing a ContigFS file and registering the obtained `mmap` area with the FlashNet RDMA device. In comparison, we also show the performance of `mmap` on `ext4` with `MAP_POPULATE` flag, and its registration cost. FlashNet can register and prepare 64GB of flash in about 90 msecs (6 msec for mmap, 84 msecs for registration). In contrast, a DRAM buffer registration

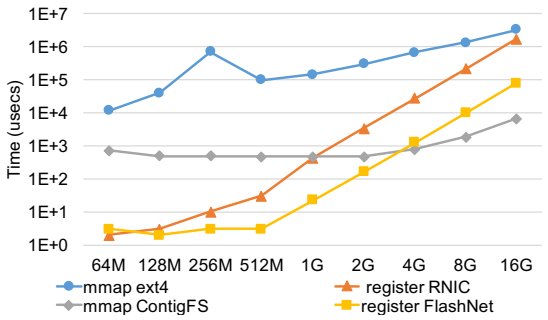

Figure 9: mmap and buffer registration cost.

takes almost 5 secs (3.3 secs for `mmap`, 1.7 secs for registration), where the majority of the time is spent in page allocation, bringing data in from the flash devices, and allocating RDMA metadata in the RNIC driver, and then installing DMA IO descriptors. FlashNet avoids this cost because it does not allocate any DRAM pages during `mmap`. DRAM pages are allocated only at the time of page faults from the flash controller managed pool of DRAM pages.

5.1.4 *Efficiency of Flash Management*

The flash controller segregates data in three levels based on their heat, age, and origin during data placement. As described earlier in Section 4.2, the `get/put` API of the controller enables it to extract write-heat information (i.e., update frequency) about a page from its usage counter. A high `get` count implies a hot data page. The data segregation and greedy GC scheme of the flash controller exhibit a significant reduction in write amplification in the common case with skew write patterns. Under Zipfian write workloads of 80/20 (80% accesses to 20% flash area) and 95/20 (95% accesses to 20% flash area), the flash controller reduces the write amplification by 38% and 83% and improves the flash lifetime by 1.6× and 5.9×, respectively, compared to a greedy-window GC scheme without data segregation.

5.1.5 *Mixed RNIC-FlashNet Deployment*

One key advantage of keeping the iWARP packet format is that FlashNet is compatible with the current iWARP RNIC hardware. In this section, we evaluate this mixed setup using Chelsio's T5 RNIC. We use an offload RDMA engine in one of the client's T5 NIC and measure the read and write latencies for a 4kB page. For a read configuration, FlashNet transmits data from a flash page to the client. For a write configuration, the T5 RNIC is used to transmit data from the client to the FlashNet server. We measured potential latency improvements from this mixed setup, which are shown in Table 3. On average, a mixed setup delivers additional 4.2% to 8.2% performance gains to FlashNet. These gains come from by-passing the full client-side software stack, and represent a realistic FlashNet deployment scenario.

5.2 Applications

We demonstrate the applicability of FlashNet using two applications: a Key-Value Store, and a distributed Sorting workload. Currently open-sourced RDMA applications e.g., HERD or RAMCloud, (without extensive modifications) are misfits for FlashNet due to two implementation-related concerns. First, these applications do not differentiate between RPC and data buffers, where both are managed by a unified

	TX-side	RX-side	4kB/poll	Gains
Read	FlashNet FlashNet	SoftiWARP T5 RNIC	54.9 μs 50.4 μs	8.2%
Write	SoftiWARP T5 RNIC	FlashNet FlashNet	72.4 μs 69.3 μs	4.2%

Table 3: 4kB IO latencies in mixed configurations.

memory manager. Naturally, high-performance RPC buffers should be kept in DRAM. Second, FlashNet currently only supports the newer OFED RDMA/CM API [61], and not the older InfiniBand API, which is used by these systems.

5.2.1 *Key-Value (KV) Store*

Key-Value (KV) stores is one of the most popular applications which is known to benefit from RDMA operations. Multiple stores have been proposed, e.g., Pilaf [52], HERD [33], FaRM [14], etc.,which explore the trade-offs and semantics in the design space. We have also implemented a distributed KV store that supports `get` and `put` operations. The main data structure is a hopscotch hash table [23] with fixed size keys and values, and a bucket size of 16; each hash table entry holds the key along with some metadata. To reduce the lookup cost, we separate the keys from the values in two tables that are indexed the same way. The whole hash table is `mmap`'ed over the capacity of a file (e.g., an `ext4` file, or a block device). Concurrent accesses to the hash table are supported with the use of the compare-and-swap primitive.

Upon a `put`, the key is hashed into a 64-bit value that is used to index the hash table, and we iterate over the hopscotch bucket until a free entry is found. When a free entry is found, the full key is copied in the table entry, and the data is copied at the value table, on the same index as the free entry. If the bucket is full, we first try to displace one of the other entries in the same bucket, and if that is unsuccessful, we push the entry into a victim bucket. On a `get`, the matching bucket is searched until the key is found; if not found, we repeat the search over the victim bucket. If the key is found in the KV store, then we read the associated data from the value table. In terms of IO amplification on random operations, the KV store has the following amortized behavior: a `put` operation will result in a read IO for the bucket and two write IOs (one for the table entry and one for the value); a `get` operation will result in two read IOs (one for the table entry and one for the value).

We implemented two network back-ends to the KV store: one over sockets and one over RDMA on FlashNet. The socket back-end uses socket operations over TCP: the control-path server runs on the storage node with 3 NVMe devices and creates a separate thread (data-path server) and socket to handle each new client connection; the client nodes create one or more threads, each creating a separate connection to the data server, and then perform data-path operations using blocking socket IO. The data-path servers operate on the shared KV hash table that has been mapped over storage. A client `get/put` involves four network operations each: a `send` and `recv` on the client and matching `recv` and `send` on the data-path server. The RDMA back-end uses one-sided RDMA operations to the KV store for the data path: the clients read and write directly to the remote flash (essentially by running the server data path locally); the storage node only spawns a control path server to `mmap` the FlashNet file and handle the client connections. A client `get` involves two remote operations: one read for reading the

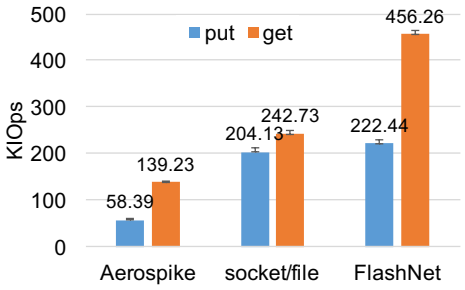

Figure 10: Performance of 4kB puts and gets.

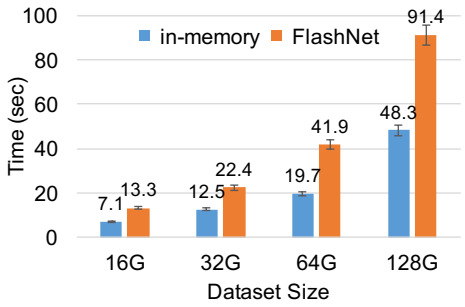

Figure 11: Distributed sorting performance.

bucket, and one read for reading the value. A `put` involves four remote operations: one read for the bucket, one RDMA compare-and-swap to lock the key, and two writes; one for value and one that updates the key and unlocks it.

We evaluate the KV store against Aerospike, a state-of-the-art NoSQL store in terms of performance and functionality [68]. We deployed Aerospike over the same storage as our KV store, using their recommended configuration (raw block device and 128K *write-block-size*). We used the Aerospike C client library benchmark to generate load for Aerospike. We performed random `puts` and `gets` of 10M keys for 4KiB key-value size; we used 4KiB to match the block size characteristics of the underlying storage and illustrate IO amplification. We drop the caches before starting each experiment. Figure 10 shows our results. For `puts`, FlashNet/KV achieves 222.4K IOPS, an improvement of 3.8× and 1.1× over Aerospike and socket/file variant, respectively. For `get` operations, the use of FlashNet's RDMA operations result in delivering 456.2K IOPS, an improvement of 3.2× and 1.8× over Aerospike and socket/file, respectively. Hence in conclusion, the potent combination of storage and RDMA operations in FlashNet API lets us develop applications that enjoy significant performance benefits over their traditional counterparts in the ways remote storage is accessed.

5.2.2 *Distributed Sorting on FlashNet*

We have modified and ported RStore [73], an RDMA-enabled distributed in-memory data store, to FlashNet. RStore uses a centralized master and distributed server/client model. In this model, servers donate parts of their DRAM to store data by preparing and registering memory buffers with RNICs. This information is then relayed to the centralized master from where the capacity allocation and distribution happens. The nodes communicate using RPC implementation on two-sided send/recv RDMA operations, and all data accesses between clients and servers happen using one-sided RDMA read/write operations.

To port RStore on FlashNet, we modified less than 100 lines (out of 15k) of code, the majority of these modifications are focused on acquiring RDMA-ready data buffers by `mmaping` ContigFS files. With these modifications in place, RStore now supports storing data on remote flash devices as files together with the currently supported DRAM buffers. No client-side changes were required and clients always accessed data transparently using one-sided RDMA operations. We evaluate RStore using one of its applications, a distributed key-value Sorter. The Sorter implements a two-phase external merge sort which reads the input data from flash files, processes it in distributed DRAM buffers, and writes out the sorted data to flash files. We run these experiments on 4 machines from our testbed where we re-distributed NVMe devices to put one in each machine. These machines run RStore data servers as well as Sorter clients. We use one additional machine to run the master.

We compare the performance of Sorter/FlashNet with its in-memory variant that read, sorts, and writes data completely in memory and uses RNIC to access data. Figure 11 shows our results. On the Y-axis is the runtime in seconds, and on the X-axis is the amount of data sorted. The key observation here is that, in comparison to the identical in-memory execution of RStore/sorter, the FlashNet version of Sorter mostly adds the time of IO from the flash devices and does not incur any additional overheads. This observation can be verified by calculating the time difference and amount of data that is read from and written to the NVMe devices. For example, in the 128GB run, the time difference between an in-memory run and over-FlashNet run is 43.1 secs, which is close to the expected NVMe device performances. With a simple calculation, we can verify that the read time should be 14.5 secs (128GB/(4 devices × 2.2 GB/sec/device)) and the write time should be 35.5 seconds (128GB/(4 devices × 0.9 GB/sec/device)). That calculation gives us a total time of 50 seconds, whereas RStore/FlashNet adds 43.1 seconds to the in-memory run. The FlashNet is faster due to bursty read/write performance being higher than what we measured in the steady state. To conclude, FlashNet (a) enables us to port an RDMA-ready system with minimal effort; and (b) imposes minimum overheads when using FlashNet's one-sided RDMA operations to access remote flash storage.

6. RELATED WORK

Inefficiencies in end-host networking stacks have first been discussed in the 1990s and have led to the design of high-performance networking stacks [76, 80, 6, 5]. RDMA networks such as InfiniBand or iWARP are the latest incarnation of these principles and are being used both in supercomputers as well as in data centers [57, 69, 52, 14, 33]. Meanwhile, commodity network and storage software stacks are constantly being optimized to eliminate inefficiencies. For example, networking efforts have focused on providing better locality [59], abstractions/APIs [22, 62], scalability [29], specialization [49, 67], per-packet processing [45, 27], etc. In parallel, the storage community has also targeted offloading [7], direct hardware access [35], specialization [64, 42], fast IO paths [66], scalability [4], interrupt coalescing [1], and polling [83, 65], etc. Many of these efforts take a very *network-alike* approach towards reducing storage overheads [74]. FlashNet is built on such common

	Remote Flash Access?	Server Involved?	Byte I/O	High-Perf. Properties	API	Limitations/Comments
`sendfile`	yes	yes	yes	no	file with socket	only deals with the TX side issues
NASD [20]	yes	no	yes	no	dist. object store	only deals with the server-side issues
SAN-level [8, 9]	yes	no	no	no	block I/O	benefits limited to the block layer
NAS-level [44, 12]	yes	no	yes	no	NFS, file I/O	apps. must copy data to/from fs bufs
BGAS [63, 17]	no	N/A	yes	yes	RDMA queues	limited to local flash access
RDMA Sys. [57, 14, 33]	no	N/A	maybe	maybe	KV, Dist. Mem.	do not involve remote storage access
FlashNet	**yes**	**no**	**yes**	**yes**	**RDMA queues**	**remote flash access using RDMA**

Table 4: Comparison of related work for networked storage access in an end-to-end manner.

high-performance IO properties from both, networking and storage, and unifies them in a single stack. In a system-wide approach, Exokernel [15, 32], and the recently proposed Arrakis [60] and IX [3] OSes contain high-performance IO stacks. These OSes eliminate overheads from the kernel by limiting its role to resource management only. Nonetheless, these isolated stack-specific efforts do not address end-to-end data transfer challenges solved by FlashNet.

Network-Attached Secure Disk (NASD) [20, 21] project eliminates many server-side CPU and OS overheads by connecting storage disks directly with the network controller. However, it relies on custom disk and networking hardware, and does not reduce overheads at the client side. BGAS [63, 17] uses RDMA, but only to access *local* flash storage. Recent work in storage disaggregation has identified network and storage protocol processing related overheads to be a performance bottleneck [36]. Naturally, there are previous works on improving performance of block-level [81, 30, 8, 41, 51], and file-level storage accesses [79, 12, 24], and even the integration of RDMA in them [10, 39, 44, 9]. Distributed file systems, such as DAFS [48, 47], DFS [13], and NFS [44], GlusterFS [19], etc., also use RDMA and present a file system interface. Commercial systems such as Violin Memory, use RDMA to access flash in an appliance setting with the SMB protocol [75]. In comparison to the aforementioned NAS/SAN approaches, FlashNet only provides a mechanism to unify storage and network processing concerns without imposing its usage. Similar to any application of RDMA API, FlashNet can be used to integrate into both SAN or NAS types of accesses. Previous efforts that tried to integrate RDMA into existing storage stacks showed limited gains due to (a) a limited scope of integration, i.e., to the block or file, not to the end-to-end application level; (b) a lack of unified concerns, for example, it is not possible to share in-kernel RDMA transport buffers of NFS with an application to avoid data copies when performing a file IO; and (c) a transparent integration of RDMA to avoid a complete re-write of the stack. However, recent efforts have demonstrated that a careful consideration is required to leverage the full potential of RDMA in native [18, 14, 33, 57] as well as JVM-based distributed systems [70, 46]. Distributed data platforms such as Crail [70] can use FlashNet's RDMA operations to transparently access remote flash using one-sided RDMA operations. And in this context, FlashNet provides a way to leverage RDMA-knowhow and apply it in the context of networked-storage to deliver the best possible performance. Applying RDMA to remote storage brings additional challenges, discussed at length by Hoefler et al. [25].

More recently, Project Donard [2] and NVMe over Fabrics (NVMeF) [54, 56] transports NVMe commands to a remote server using RDMA and can even transfer data to an RDMA device directly using peer-to-peer PCI transactions. These efforts share many key properties and design principles with FlashNet, which validates our choices. However, by using the RDMA verbs API, FlashNet exposes a generic, device independent interface with the richer RDMA semantics (one-sided operations, byte-addressability, asynchronous IO, etc.) for remote storage access, whereas NVMe and hence NVMeF as well currently only supports block-level device access. The current FlashNet design also includes a file system design, hence offering a NAS-level data sharing. Furthermore, FlashNet abstracts all device specific management operations and leaves device management as an orthogonal task. Since FlashNet keeps the media access protocol local to the storage device, it omits the need to introduce another wire protocol for each new class of storage devices.

Like FlashNet, ReFlex [37] takes a holistic approach towards optimizing network as well as storage IO for remote flash accesses. It proposes a server design that is integrated with the IX OS for the best performance in terms of QoS, isolation, and efficiency. In comparison, FlashNet aims to extend RDMA operations on commodity OSes to access remote flash storage, and leaves application to use these operations as they see fit. FlashNet's on-demand memory pinning can be augmented with the recently proposed page fault support from RDMA NICs [43]. However, FlashNet's on-demand mechanism is also used to track heat and access information about flash pages, which is then used to supplement the garbage collection for superior flash management.

7. CONCLUSION

In this paper, we have presented FlashNet, a unified software IO stack that provides direct, high-performance access to remote flash storage. The unified stack delivered 1.22M IOPS to clients, which is only limited by the network performance. FlashNet achieves this performance by adopting the well-known path separation principle from RDMA networks and extending it to storage by co-designing a flash controller and a file system with it. As a demonstration of FlashNet's capabilities and its API, we have developed a Key-Value store, and ported an existing RDMA-ready distributed sorting application on it. They both perform well. For example, the performance of the KV store is nearly doubled from 242K to 456K IOPS, and the Sorter experienced minimal overheads when using FlashNet's one-sided RDMA operations. As RDMA networking and API gain wide-spread usage, FlashNet opens the door to integrate storage as a first-class citizen in the high-performance IO hierarchy, and unifies efforts across the network and storage stacks.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Edward Bortnikov, for their helpful reviews and comments.

9. REFERENCES

[1] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC '11, pages 45–58, 2011.

[2] S. Bates. Donard: NVM Express for Peer-2-Peer between SSDs and other PCIe Devices, http://www.snia.org/sites/default/files/SDC15_presentations/nvme_fab/StephenBates_Donard_NVM_Express_Peer-2_Peer.pdf.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, 2014.

[4] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, 2013.

[5] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 142–153, 1994.

[6] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 245–259, 1996.

[7] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. 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, pages 387–400, 2012.

[8] A. M. Caulfield and S. Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 464–474, 2013.

[9] M. Chadalapaka, H. Shah, U. Elzur, P. Thaler, and M. Ko. A Study of iSCSI Extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 209–219, 2003.

[10] L. Chai, X. Ouyang, R. Noronha, and D. K. Panda. pNFS/PVFS2 over InfiniBand: Early Experiences. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 5–11, 2007.

[11] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 3(4):837–863, Nov. 2004.

[12] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: Scalable High-performance Storage on Virtualized Non-volatile Memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 17–31, 2014.

[13] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST'03, pages 175–188, 2003.

[14] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, 2014.

[15] D. R. Engler, M. F. Kaashoek, , and J. O. Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[16] Flexible I/O tester, https://linux.die.net/man/1/fio.

[17] Fitch, Blake G. and others. Blue Gene Active Storage (BGAS) for High Performance BG/Q I/O and Scalable Data-centric Analytics, https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/bgas/bgas-fitch.pdf?__blob=publicationFile.

[18] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 553–560, 2009.

[19] GlusterFS, http://www.gluster.org/.

[20] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-effective, High-bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 92–103, 1998.

[21] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File Server Scaling with Network-attached Secure Disks. *SIGMETRICS*, 25(1):272–284, June 1997.

[22] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, 2012.

[23] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC '08, pages 350–364, 2008.

[24] D. Hildebrand and P. Honeyman. Exporting Storage Systems in a Scalable Manner with pNFS. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, pages 18–27, 2005.

[25] T. Hoefler, R. B. Ross, and T. Roscoe. Distributing the Data Plane for Remote Storage Access. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, May 2015.

[26] X.-Y. Hu, R. Haas, and E. Eleftheriou. Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 237–247, July 2011.

[27] Intel. DPDK: Data Plane Development Kit, http://dpdk.org/.

[28] N. Ioannou, I. Koltsidas, R. Pletka, S. Tomic, R. Stoica, T. Weigold, and E. Eleftheriou. SALSA: Treating the Weaknesses of Low-cost Flash in Software. In *as a poster in 6th Annual Non-Volatile Memories Workshop*, 2015.

[29] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[30] A. Joglekar, M. E. Kounavis, and F. L. Berry. A Scalable and High Performance Software iSCSI Implementation. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, FAST'05, pages 267–280, 2005.

[31] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 85–100, 2010.

[32] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, 1997.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, 2014.

[34] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, 2016.

[35] H.-J. Kim, Y.-S. Lee, and J.-S. 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.

[36] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[37] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash == Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, 2017.

[38] K. C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965.

[39] E. Koukis, A. Nanos, and N. Koziris. GMBlock: Optimizing data movement in a block-level storage sharing system over Myrinet. *Cluster Computing*, 13(4):349–372, 2010.

[40] 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, 2015.

[41] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, 1996.

[42] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and A. Arvind. Application-managed Flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 339–353, 2016.

[43] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 449–466, 2017.

[44] B. Li, P. Zhang, Z. Huo, and D. Meng. Early Experiences with Write-Write Design of NFS over RDMA. In *IEEE International Conference on Networking, Architecture, and Storage.*, pages 303–308, July 2009.

[45] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, 2014.

[46] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance Design of Apache Spark with RDMA and its Benefits on Various Workloads. In *IEEE International Conference on Big Data*, pages 253–262, 2016.

[47] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most Out of Direct-Access Network Attached Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 189–202, 2003.

[48] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of the 2002 USENIX ATC*, pages 1–14, 2002.

[49] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, 2014.

[50] B. Metzler et al. SoftiWARP: Software iWARP kernel driver and user library for Linux at https://github.com/zrlio/softiwarp, accessed February, 2017.

[51] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and

O. Khan. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 257–273, 2014.

[52] C. Mitchell, Y. Geng, and J. Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, 2013.

[53] Netperf: A network performance benchmark. http://www.netperf.org.

[54] NVM Express over Fabrics Specification 1.0, http://www.nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605-1.pdf.

[55] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. Non-volatile Storage. *Queue*, 13(9):20:33–20:56, Nov. 2015.

[56] W. Noureddine. Implementing NVMe over Fabrics, http://www.snia.org/sites/default/files/SDC15_presentations/networking/WaelNoureddine_Implementing_%20NVMe_revision.pdf.

[57] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[58] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 15–28, 1999.

[59] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, 2012.

[60] 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, 2014.

[61] RDMA communication manager API, https://linux.die.net/man/7/rdma_cm.

[62] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 101–112, 2012.

[63] F. Schürmann et al. Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 331–347. Springer International Publishing, 2014.

[64] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, 2014.

[65] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom. Dynamic Interval Polling and Pipelined Post I/O Processing for Low-latency Storage Class Memory. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, 2013.

[66] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 483–488, 2014.

[67] Solarflare Communications Inc. OpenOnload at http://www.openonload.org/, 2013.

[68] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic. Aerospike: Architecture of a Real-time Operational DBMS. *Proc. VLDB Endow.*, pages 1389–1400, 2016.

[69] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-sided Operations in soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 347–353, 2012.

[70] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 40(1):40–52, March 2017.

[71] N. Talagala. Native Flash Support for Applications, at Flash Memory Summit http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120823_S304B_Talagala.pdf, 2012.

[72] A. Trivedi, B. Metzler, and P. Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the 2nd APSys*, pages 17:1–17:5, 2011.

[73] A. Trivedi, P. Stuedi, B. Metzler, C. Lutz, M. Schmatz, and T. R. Gross. RStore: A Direct-Access DRAM-based Data Store. In *35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 674–685, June 2015.

[74] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross. Unified High-Performance I/O: One Stack to Rule Them All. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[75] Violin and Microsoft's High-Performance, All-Flash Enterprise Storage, http://www.violin-memory.com/blog/violin-and-microsoft-windows-flash-array/2/.

[76] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, 1995.

[77] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, 2015.

[78] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ANViL: Advanced Virtualization for

Modern Non-volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 111–118, 2015.

[79] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, 2008.

[80] J. Wilkes. Hamlyn - an Interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, 1992.

[81] D. Xinidis, A. Bilas, and M. D. Flouris. Performance Evaluation of Commodity iSCSI-Based Storage Systems. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, pages 261–269. IEEE Computer Society, 2005.

[82] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 6:1–6:11, 2015.

[83] J. Yang, D. B. Minturn, and F. Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 25–32, 2012.