FlashNet: Flash/Network Stack Co-Design

ANIMESH TRIVEDI, NIKOLAS IOANNOU, BERNARD METZLER, PATRICK STUEDI, JONAS PFEFFERLE, and KORNILIOS KOURTIS, IBM Research, Zurich, Switzerland
IOANNIS KOLTSIDAS, Google
THOMAS R. GROSS, ETH Zurich, Switzerland

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 I/O 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 article, we present FlashNet, a software I/O 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 I/O efficiency. In micro-benchmarks, FlashNet improves 4kB network I/O operations per second (IOPS by 38.6% to 1.22M, decreases access latency by 43.5% to 50.4μs, 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**;

Additional Key Words and Phrases: RDMA, flash, network storage, performance, operating systems

ACM Reference format:
Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Kornilios Kourtis, Ioannis Koltsidas, and Thomas R. Gross. 2018. FlashNet: Flash/Network Stack Co-Design. *ACM Trans. Storage* 14, 4, Article 30 (December 2018), 29 pages.
https://doi.org/10.1145/3239562

Authors' addresses: A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, and K. Kourtis, IBM Research, Saeumerstrasse 4, Zurich 8803, Switzerland; emails: {atr, nio, bmt, stu, jpf, kou}@zurich.ibm.com; I. Koltsidas, Google, Brandschenkestrasse 110, 8002 Zurich, Switzerland; email: iko@zurich.ibm.com; T. R. Gross, ETH Zurich, Computer Science Department, Zurich, 8092, Switzerland; email: thomas.gross@inf.ethz.ch.

ACM 1553-3077/2018/12-ART30 $15.00
https://doi.org/10.1145/3239562

1 INTRODUCTION

Modern distributed applications such as data processing stacks (Spark or MapReduce), web services, databases, and so on, routinely store, access, and analyze Terabytes of data stored across hundreds of storage servers. Consequently, their performance depends considerably on the I/O performance of the many storage *and* network devices involved. Thankfully, the I/O performance of network and storage devices has been dramatically improved over the past decade. Ethernet, the most popular interconnect technology, has evolved from 1 to 10, 40, and now 100Gbps data rates with single-digit microsecond link latencies. Flash storage has delivered two to four orders of magnitude bandwidth and latency improvements over HDDs, while recent advancements in non-volatile memory technologies [40, 68] promise to improve performance further by one or more orders of magnitude.

However, translating these raw I/O performance improvements into application-level gains remains a challenge due to multiple factors. First, traditional I/O stacks[1] are designed assuming *slow* I/O and *fast* CPUs, which is no longer true [5]. Consequently, the cost of maintaining generic OS abstractions, I/O interfaces, scheduling, context switches, implementation-related inefficiencies, and so on, contributes significantly to the loss of I/O efficiency. Second, recent improvement efforts either target the network [29, 42, 64] or the storage stack [8, 50, 81, 101] 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 while engaging the operating system many times for each I/O operation. Last, many established solutions in this space, such as NFS or iSCSI, aim to deliver data only up to the file (Network Attach Storage (NAS)) or block (Storage Area Network (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 article, we present FlashNet, a co-designed I/O stack that delivers high-performance data access to networked clients. The FlashNet stack builds on the data and control path separation philosophy of commodity Remote Direct Memory Access or RDMA networks. RDMA networks, e.g., InfiniBand, iWARP, or RoCE, and so on, have already shown to deliver high network performance to various applications [19, 48, 67, 70, 83], and related concepts [11, 100] and even APIs [91] are explored in the storage domain as well. In FlashNet, we unify these fragmented efforts across the I/O stacks and extend the path separation philosophy of RDMA networks to storage with the help of a co-designed flash file 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 (on both the client and the server sides), 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 I/O 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, and so on. As a result, FlashNet delivers 1.22M I/O operations per second (IOPS) (100% of 40Gbps network bandwidth performance), decreases access latencies by 43.5% to 50.4μs, and improves the flash lifetime by 1.6-5.9× for writes.

FlashNet is designed to be fully RDMA compatible, thus enabling hybrid RNIC/FlashNet deployments. Consequently, applications that have previously used RDMA to efficiently access remote memory require minimum changes to access data to/from flash storage using FlashNet. To

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

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 dis-aggregated 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 [51]. 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 [90], which uses RDMA to access data from remote DRAMs. We have ported RStore 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 runtime, and the ported Sorter delivers a performance that is the sum of its in-memory runtime 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 I/O stack that consists of an 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 I/O 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 7) 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 IOPS delivered by the server for the following configurations:

- *iSCSI:* iSCSI is the de-facto block-level data access protocol in data centers. It is configured with in-kernel iSCSI drivers. iSCSI targets are the three devices that were partitioned and distributed between clients. The iSCSI configuration is tuned as described by Klimovic et al. [51].
- *NFS:* NFS is a popular network file system protocol. In our experiments, the NFS server runs in kernel and accesses data from the `ext4` file system mounted on top of the software RAID-0 device made on top of the three NVMe devices.
- *Key-Value:* We evaluate the performance of Aerospike KV, a state-of-the-art NoSQL store in terms of performance and functionality [82]. Aerospike is configured to run over the raw block device in the recommended high-performance configuration (see Section 7.2.1).
- *Hadoop Distributed File System (HDFS):* And, last, we measure the Apache Hadoop distributed file system (HDFS) [97]. Here, HDFS clients access a 4kB block randomly in a large, private 8GB file.

We generate load for file and block I/O with `fio` [3]. We include performance numbers from the device specification marked as `spec` and local `block`-level I/O 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

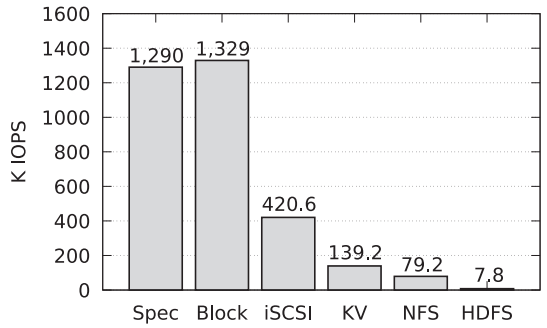

Fig. 1. Random 4kB read IOPS performance under various systems and configurations. Spec shows the device performance from the specification. Block is the measured performance at the local block I/O level.

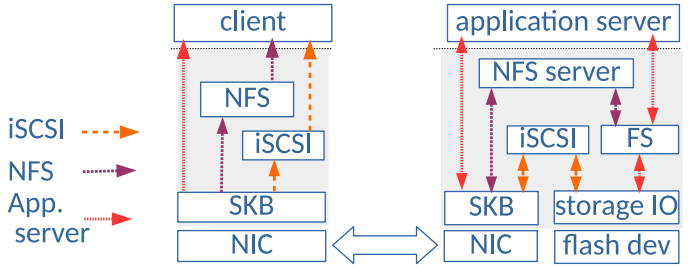

Fig. 2. I/O paths taken by data in tested settings.

limited by the CPU performance. In Figure 2, we illustrate various I/O paths involved between a server and a client. Along these lines, the CPU has to to do network processing, I/O buffer management, file system execution, block I/O, scheduling, and execute generic OS services and abstractions. In line with previous findings [51], 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 dataflow. The cost of scheduling, context switches, cache flushes, I/O resource management (as they are tied to the process abstraction), and so on, becomes an overhead. And, last, 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 operations has been proposed to completely eliminate OS and servers from an application design, e.g., KV stores [19, 67, 83], distributed transactions [20, 49], locking [94], and so on. 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 [11, 57]. FlashNet is built on similar principles but goes a step further by unifying these isolated efforts and building a holistic, end-to-end data path for fast remote data accesses.

2.1 Background: RDMA Primer

Before proceeding further with the design and implementation of FlashNet, in this section we provide relevant background information on the RDMA technology. RDMA is a userspace

networking technology that provides high performance by eliminating unnecessary OS and application involvement (and associated overhead) from the network I/O processing path. One of the key principles of RDMA is the explicit separation of control operations to setup network resources, from data operations triggering the actual data transfer. Examples of networking resources are transmission (TX) and receiving (RX) queues (collectively called Queue Pairs (QPs)), completion notification queues (CQs), and network buffers. Those resources need to be pre-allocated and applications are given direct userspace access to them with the help of the operating system. Network buffers are registered with the RDMA network controller to generate a valid buffer identifier tag also known as the Steering Tag (STag). These STags are used to resolve RX and TX buffer locations associated with an RDMA network operation. Hence, in the network request processing path there are no stalls for scheduling and/or resource allocation.

RDMA employs a message oriented protocol where a network I/O request issued by an application makes an RDMA message. An application is only notified when a complete message is received or transmitted, thus reducing its involvement in network packet processing. A single RDMA message is broken down into various fragments. Each fragment is delivered and processed in order. The RDMA header of each fragment contains the STags of the involved I/O buffers and a fragment offset that is used to identify if it is the last fragment of the message. The RDMA networking API goes beyond simple send and receive operations. One-sided RDMA operations allow a client to directly read or write data from a prepared remote network buffer without involving the remote server application. Because of its design and its minimum application involvement, one-sided RDMA operations have the lowest latency and the highest bandwidth. A comprehensive introduction to RDMA can be found in Reference [24].

Though traditionally implemented in hardware as an offloaded network controller, RDMA concepts and ideas can be implemented in software as well. Multiple open-sourced implementations are available today [65, 88]. Naturally, a software-only implementation cannot offer the same level of performance as an offloaded hardware. However, these devices are shown to be useful to improve CPU efficiency and performance in circumstances where the RDMA hardware might not be available [89]. A hybrid software-hardware solution is also possible, either in the form of indirection [92] or as a paravirtualized RDMA device [74].

3 DESIGN OF FLASHNET

The high-level goal of FlashNet is to improve I/O efficiency (and, consequently, performance) of data-intensive applications when accessing remote storage devices. Inspired by the fact that the evolution of modern userspace storage stacks and high-performance userspace networking stacks share many key performance properties, application interfaces, and design goals [91], FlashNet uses the RDMA principle, semantics, and associated mechanisms to achieve its performance goals. For example, it pre-allocates networking and storage resources ahead of data accesses. Using the same one-sided RDMA operations, FlashNet eliminates the server application's involvement from the data processing path.

3.1 Overview

FlashNet is a unified *software* stack that consists of three logical components: a flash controller, a file system (ContigFS), and an RDMA controller.[2] By co-developing these components, we design a unified, *end-to-end* data path where data can flow between a remote flash device and a client buffer

[2]FlashNet is a software stack, though, a hardware prototype would be possible with the help from the network (e.g., Reference [58]) and storage device controllers. We use separate controller names to highlight their roles in the overall FlashNet architecture.

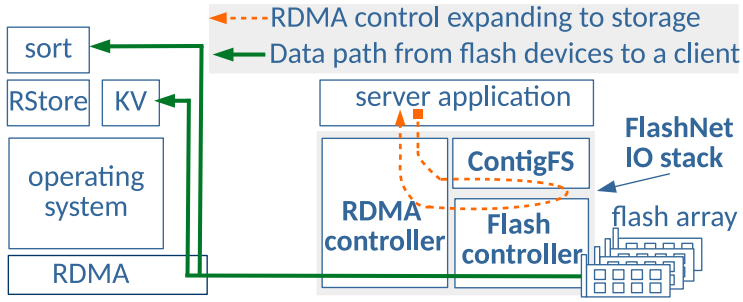

Fig. 3. FlashNet stack illustrating the network-storage setup (dotted, red line) and the end-to-end dataflow path (bold green line).

uninterrupted. 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 I/O flows:* FlashNet leverages the path separation philosophy of RDMA networks and extends it to storage devices to completely eliminate an application's involvement from data transfer orchestration in an end-to-end manner.
(2) *Minimize system overheads:* FlashNet eliminates the file system and much of the generic OS and the VFS code from the extended data transfer path by designing a file system that uses a simple file layout. Hence, the file system and associated actions (e.g., inode look-ups, checks, location translations, etc.) are eliminated from fast data paths between the network and storage controllers.
(3) *Keep application interface simple and clean:* With the help of the file system, FlashNet leverages the basic ideas of files and `mmap`-based I/O to autonomously and efficiently manage data transfers over RDMA networks. Using existing standard mechanisms, FlashNet plugs storage buffers into the application address space and lets the application manage them in a unified manner with other DRAM buffers.

3.2 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 that could be useful to absorb the bursty, concurrent nature of network I/O, 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 to 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 [46]. A simpler file layout enables removing the file system from network data transfers. And, last, 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 [46, 95]. The FlashNet flash controller is implemented using the SALSA software-defined storage framework [41]. SALSA splits the available storage space (spanning one or more storage devices) into *segments* (e.g., 1GB), and segments are, in turn, split into *grains* (e.g., 4kB). SALSA supports multiple "controllers" over a single pool of storage, each with its own policy and allocation streams. A segment can only be owned by a single controller at any point in time. SALSA provides space allocation API for allocating grains, invalidating grains, as well as a garbage collection callback that upcalls the owning controller to relocate a consecutive range of grains. For more details about the SALSA architecture please refer to Reference [41].

FlashNet implements its flash management as a SALSA controller. 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 are 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 an asynchronous `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 are 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 generalized version of the greed [15] and circular buffer [76] policies, augmented with an aging factor. This aging factor improves the performance of the algorithm under a skewed write workload without negatively impacting its performance under random writes. It uses 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 [34]. Ideally, a number of data streams equal to the update frequencies that the workload exhibits should be chosen. In practice, however, the supported number of data streams is limited by hardware or metadata resources. Our controller performs a three-level data segregation scheme based on (a) the logical origin of data blocks, (b) their age in the system, and (c) their frequency and heat of updates tracked with the `get/put` API. A detailed discussion of the GC algorithm and the data placement policies is outside the scope of this article and is provided elsewhere [22].

To protect against crashes, the flash controller logs updates to the mapping table (LBA-to-PBA). On 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, then the mapping table is fully restored from a fixed location. Otherwise, the mapping table is reconstructed by scanning all the log metadata pages, based on back-pointers and timestamps.

3.3 Contiguous File System (ContigFS)

File systems such as `ext4`, are one of the most popular ways of storing and organizing data. In this work, we want to keep the benefits of using a file system for data storage management while

delivering the full performance of storage devices to clients over the network. Unfortunately, as storage devices get faster, the constant and unnecessary involvement of a file system in every aspect of I/O (local or networked) operations generates a significant amount of overhead [11, 55]. In the I/O 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 I/O 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 simple translation can also be done by the network controller, hence potentially removing the file system and application involvement from the networked I/O. Another alternative that can achieve a simple, range-based offset translation is raw block-device interface. In this setup, a block device can be used to store data using contiguous storage blocks. However, this option eliminates many applications that use file system interface and properties (e.g., hierarchical naming and access control) from running on FlashNet. Hence, we decided to develop a simple, range-based file system that is optimized for fast I/O operations.

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 are staged for access and dirty data are 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 I/O-lite system [71], 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 by reference counting.

3.4 The RDMA Controller

The RDMA controller of FlashNet extends the data and control paths [93, 98] of commodity 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 dataflow path. To achieve this, FlashNet translates necessary abstractions in advance from an 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 pre-registered with the network controller to generate an RDMA buffer identifier tag or STag. This STag is used in subsequent RDMA operations to identify network source or sink buffers without involving the application to steer dataflows. FlashNet uses the same mechanism to identify files and offsets to resolve data locations that 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

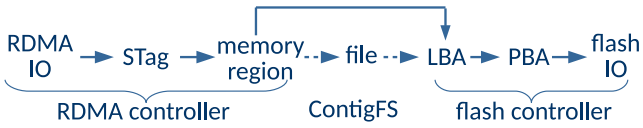

Fig. 4. Abstraction translations inside FlashNet. A fast translation from a memory region to an LBA is accomplished with the use of memory-mapped I/O (i.e., mmap) and buffer registration mechanisms.

data path, the file system is eliminated and the two device controllers talk to each other to manage dataflows. Figure 4 shows the end-to-end translation process between these abstractions on the extended control and the data path.

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, and (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.

The use of the RDMA further brings byte granular, low-latency, high-bandwidth flash accesses into the Remote Memory Access programming model [33]. Many of its key performance properties such as userspace-mapped I/O queues, batching, asynchronous I/O, polling, and so on, have been explored and shown to be useful in the context of local flash accesses as well [11, 80, 101]. FlashNet provides a high-performance I/O interface around these unified RDMA properties [91]. This holistic approach collapses the layer structure and thus, efficiency is gained by creating fast, non-blocking, asynchronous end-to-end dataflow paths [7].

3.5 The Life of an I/O Operation

To demonstrate how various components come together, we illustrate an example where a server serves data to a networked client using FlashNet's one-sided RDMA read operation in Figure 5. Section 5 presents the pseudo source code for this example and discusses further application-level considerations when using FlashNet. A server application starts by mmaping a file (step ❶) that server wants to serve to network clients. It then registers the mmap address with the FlashNet RDMA controller to prepare the mmap region 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. On receiving an incoming RDMA read request, the RDMA controller first resolves the flash buffer using the STag present in the request.

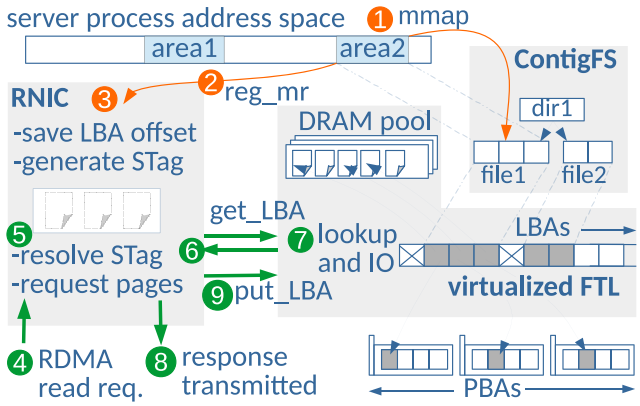

Fig. 5. The life of a FlashNet RDMA read. The orange/thin lines represent the control setup steps (1–3), whereas green/thick lines represent the dataflow steps (4–9).

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 (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). On 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 are 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 as kernel modules in the Linux kernel and do not rely on any specific hardware support. The implementation of the in-kernel virtualized flash controller is based on the SALSA software-defined storage framework [41]. The controller is extended to support the in-place, DRAM-based page sharing and callback based on a non-blocking I/O API (see Table 1). In order to support local accesses (Section 3.3), the controller also implements the block interface, utilizing the device-mapper (DM) framework. The block interface is implemented on top of the same non-blocking I/O 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 [65, 89] that provides the complete iWARP RDMA features and semantics. It uses memory-mapped I/O queues and non-blocking kernel TCP sockets with per-core kernel threads to perform asynchronous network I/O 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 hybrid hardware-software deployments. The whole FlashNet framework does not require any changes to the kernel and network or storage drivers.

4.1 File Management

ContigFS file management is similar to the Direct File System [46, 85]. 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

Table 1. The Flash Controller (Abridged) API

API functions	Description
is_contigfs_vma(va, len)	is passed vma segment a ContigFS mmap'ed area
get_LBA(addr, len, flags, callback*)	gets DRAM pool pages for I/O
put_LBA(addr, len, flags, callback*)	puts pool pages in a LBA range

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 [53].

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.

4.2 Flash Page and Buffer Management

In this section, we describe FlashNet's page pool management stragegy when processing I/O requests. DRAM pages from this pool are used to buffer and share data from flash devices. The pool 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 idea of the interface is to implement an access reference counting mechanism for an efficient data staging and buffering. 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 that 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 on 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`.

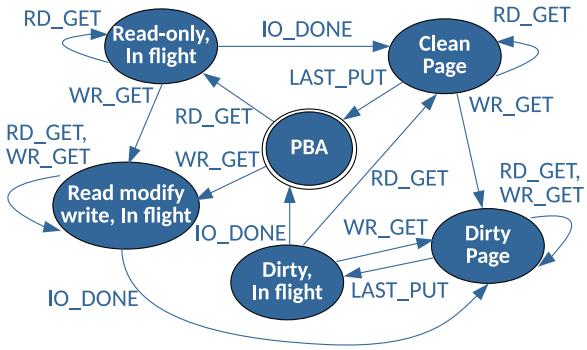

Fig. 6. The state machine of a flash LBA page.

Additionally, for dirty pages, a call to `put_LBA` can optionally generate a callback when the LBA 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

Figure 7 shows the concurrency and page sharing model of FlashNet. 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. On 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 are 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 3.4, 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, then 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. 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

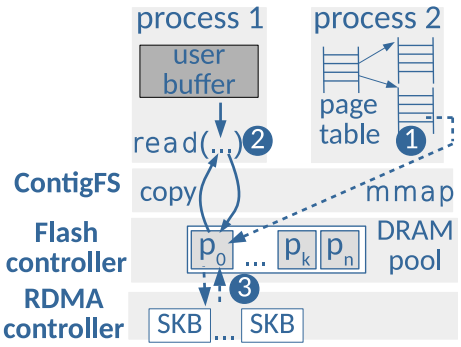

Fig. 7. Semantics of a page sharing between (1) mmap, (2) POSIX I/O, (3) remote RDMA accesses.

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 a 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, on 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 APPLICATION API AND EXAMPLE

In this section, we show an example how a server application can use FlashNet to integrate flash storage accesses with RDMA operations. As discussed previously, FlashNet combines the RDMA interface from network to the `mmap` abstraction of storage stack to build a unified end-to-end stack. Instead of providing a comprehensive introduction to the RDMA programming (available here [24, 87]), we are going to focus on key mechanisms that allow the integration of FlashNet to an already RDMA-aware server application. FlashNet's RDMA controller device is implemented as a software RDMA device in the Open-Fabrics Alliance (OFA) RDMA framework. A sample output of `ibv_devices` is shown in listing 1. This listing demonstrate that software RDMA devices of FlashNet (shown as `flashnet_lo` and `flashnet_eno3`) coexist together with a hardware RDMA device of Mellanox (shown as `mlx4_0`). From a client-side, there is no observable difference between communicating to a FlashNet's software RDMA device or an actual hardware RDMA device.

The abstraction that links a storage device to a FlashNet RDMA device is the virtual address space. A skeleton code how a server application can achieve this is shown in listing 2. The code

```
user@machine1:~$ ibv_devices
    device                 node GUID
    ------              ----------------
    flashnet_lo         7369775f6c6f0000
    flashnet_eno3       0894ef0762ea0000
    mlx4_0              e41d2d0300748260
user@machine1:~$
```

Listing 1. A sample output of ibv_device command showing two FlashNet devices and one Mellanox device.

```
1  int fd, mode = ..., flags = ..., size = 1 << 20; // 1MB
2  /* We assume that ContigFS is already mounted at "/mnt/contigfs/". The server application either
3  * creates a new file with a finite size, or open an already existing file.
4  */
5  if (flags & O_CREAT ) {
6    /* create a new file and set a size to it */
7    fd = open("/mnt/contigfs/newFile", flags, mode);
8    fallocate(fd, mode /*permissions*/, 0 /*zero offset*/, size /*size*/));
9    /* the same mechanism can be used to increase the capacity of the file */
10 } else {
11   /* open an existing file to serve its content */
12   fd = open("/mnt/contigfs/oldFile", flags);
13 }
14
15 /* mmap the file into the process address space */
16 void *faddr = mmap(NULL, size, prot, flags, fd, off);
17
18 /* register the mmap address with the RDMA device. We assume that a protection domain object (pd)
19 * can be acquired by either opening the RDMA device directly or from an established connection.
20 * After acquiring the address from a ContigFS mount point, the mmap address can be used freely with
21 * any RDMA code. However, the application must know that the data is accessed from flash devices.
22 */
23 struct ibv_mr *flash_mr = ibv_reg_mr(pd, faddr, size, access);
24
25 /* initialize an RDMA operation using the flash memory region object "flash_mr" */
26 struct ibv_send_wr rdma_msg={...};
27
28 /* post the rdma operation on an QP for I/O operations */
29 ibv_post_send(qp, &rdma_msg,...);
```

Listing 2. Skeleton example code of acquiring memory for RDMA operations using FlashNet. A comprehensive error handling code is omitted.

example assumes that ContigFS is already mounted at a pre-defined location (e.g., `/mnt/contigfs/`). As ContigFS hooks in the VFS layer of Linux, this can be achieved by the standard `mount` command. After setting the right permission, applications can either create new files or read/write already existing file at the mounted location (lines 5–13). To pre-allocate a capacity for a newly created file, the example application calls `fallocate` call. The same mechanism can be used to expand the capacity of a file. Otherwise, using the standard POSIX `write` call also increases the size of the file. With the right size and offset, the file is then `mmaped` in the application address space (line 16), and a valid address (`void *faddr`) is acquired. This address is then registered with the RDMA device using the `ibv_reg_mr` call (line 23). From, this step onwards, this address can be

used like any other virtual memory address (lines 24–29). RDMA operations in this address range will be served by the FlashNet's RDMA device. There are no changes required on the client side.

There are three practical issues that application must consider while integrating FlashNet to server data files from flash over RDMA operations. First, often RDMA-based systems implement application-specific, memory/buffer management logic. The idea here is to pre-allocate and register these buffers, and reuse them for multiple RDMA operations to amortize the cost of the slow memory-allocation operation. These buffers are also used for the send/recv (two-sided) based communication, for example, RPC calls for synchronization. Naturally, applications want to have their high-performance RPC buffers in DRAM and data buffers on flash. Hence, an application must be aware of the location of a buffer, and use it accordingly. All RDMA-ready applications do not make this distinction, because RDMA operations so far are only destined for DRAM. Second, FlashNet only offers OFED RDMA/CM API [1] to applications. The native InfiniBand API is not supported at this moment. Third, the RDMA controller of FlashNet implements the iWARP protocol in software. Hence, the current FlashNet prototype can only operate with other iWARP RDMA stacks (either hardware or software). Because of these three limitations, currently open-sourced RDMA applications, e.g., HERD or RAMCloud (without extensive modifications), are misfits for FlashNet. Hence, for our application-level evaluation (Section 7.2), we build one native and port one application to FlashNet. The native application is a key-value store. The ported application is a distributed in-memory data store [90], which already makes the distinction between RPC and data buffers.

6 FUTURE DIRECTIONS

Persistent RDMA Credentials: RDMA semantics are currently defined within the scope of a process. When a process dies, the associated RDMA resources are destroyed. However, with the flash integration, FlashNet brings persistency to RDMA operations. Hence, persistent storage services built on FlashNet, such as a filesystem, may require persistent RDMA credentials. Furthermore, a client might have cached RDMA credentials for one-sided operations. We are investigating, first, whether it makes sense to to preserve RDMA credentials and, second, if so, then how can we preserve or re-generate (doable by replaying an operational log) RDMA credentials (registered flash regions, STags, mapped virtual address ranges, etc.) across a process or the system restart.

Current RDMA semantics: The current FlashNet prototype is bound to the RDMA protocol (specifically iWARP) and semantics. RDMA up to now, however, is defined for DRAM only. Consequently, the current RDMA specification does not provide any ordering, consistency, and synchronization guarantees in the presence of concurrent accesses or failures. In a limited scope, atomic operations with in a single RDMA device, fences for ordering guarantees, and so on, are discussed. However, there is no comprehensive treatment of these issues specially in presence of out-of-order completion/notification by NVMe devices and failures. There have been some initial proposals for extensions in the current RDMA semantics to include commits and durability operations for byte-addressable remote persistent memories [18, 86].

Quality-of-Service for FlashNet: Achieving isolation among various RDMA connections requires applications to carefully tune a variety of device and network parameters [102]. Klimovic et al. make a similar case for delivering "predictable performance" for remote flash accesses [52]. Providing fairness, isolation, and good quality-of-service on the end-to-end data path of FlashNet is a focus of future efforts.

7 EVALUATION

We evaluate FlashNet on a cluster of 17 machines with dual Xeon E5-2690 CPUs, 256GB 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

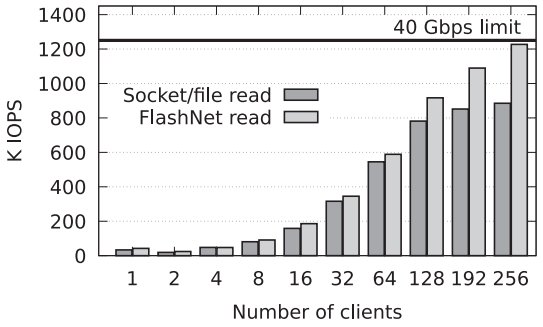

Fig. 8. 4kB random read performance.

machines run clients that repeatedly request the server for data stored in flash. Clients use SoftiWARP as a software RDMA device [65]. All numbers reported here are the average of of three runs, each lasting 60s. FlashNet experiments are done with the files on ContigFS while other configurations use the `ext4`. The key performance highlights are as follows:

- 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 40Gbps network with 1.22M 4kB IOPS, the network maximum possible on our 40Gbps link.
- The direct end-to-end data path of FlashNet helps to reduce the network read latency of a 4kB flash page from 89.2μs to 54.9μs, a 38.4% improvement. This latency can be reduced further by a margin of 8.2% to 50.4μs 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.

7.1 Micro-Benchmarks

We first show a set of micro-benchmarks to highlight the performance efficiency of FlashNet in a conventional server-client setup. Our benchmarks are implemented in `netperf` [45] with two modifications: (1) support for RDMA operations and (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 I/O operations while highlighting the potential raw performances.

7.1.1 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 is evident, both approaches scale well with the number of clients. The socket/file-based approach stops scaling linearly around

Table 2. Breakdown of CPU Cycles

	network	storage	I/O drivers	scheduling	kernel	app-logic	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%

The key performance gains of FlashNet comes from saving the cycles in scheduling, storage, and spending more time in I/O logic processing logic routines.

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 that is saturated at 40Gbps when delivering 1.22M 4kB IOPS. Hence, FlashNet reaches its first objective of delivering full storage performance across the network.

Table 2 breaks down the CPU cycles into network, storage, I/O device drivers, scheduling, kernel, request processing/application logic, 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, and so on. Most of these routines are executed for *every* network-storage request. Furthermore, the traditional I/O stack architecture experiences high context switching and scheduling related overheads. This is because most of the I/O resources (e.g., network sockets, buffers, etc.) are tied to the process abstraction. These resources need to be valid during the I/O processing and hence require the application process to be scheduled for I/O 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 I/O path and request processing does not require server process scheduling. The CPU cycle gains from here are then used in the request processing.

7.1.2 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 59.8μs and 83.3μs, respectively. In comparison, the socket/file-based client takes 89.2μs and 114.9μs, respectively. The read performance numbers can be further improved to 54.9μs 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 17.4Gbps of read and write bandwidth, respectively. In comparison, FlashNet delivers 24.4Gbps (gains: 79.4%) and 25.6Gbps (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 40Gbps bandwidths to their clients.

7.1.3 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 [25]. 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 `mmaping` 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 16GB of flash in about 90ms (6ms for mmap, 84ms for registration). In contrast, a DRAM buffer registration takes almost 5s (3.3s for `mmap` and 1.7s for

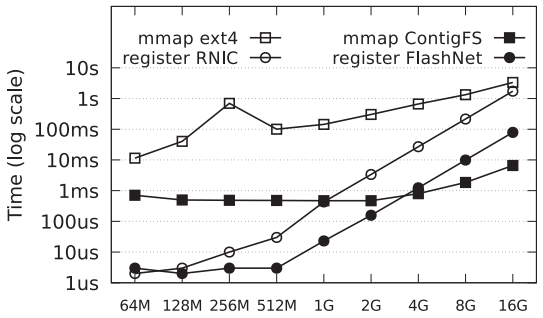

Fig. 9. mmap and buffer registration cost.

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 I/O 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. However, during registration on FlashNet, all necessary metadata are allocated.

7.1.4 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 GC algorithm of the flash controller exhibit a significant reduction in write amplification in the common case of skew write patterns, compared to existing GC schemes. 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.

7.1.5 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.

7.1.6 Next-Generation of NVMe Storage. In this section, we present the latency evaluation of FlashNet on an Intel Optane SSD 900P Series storage device [39]. The Optane technology represents the next generation of NVM storage with a superior performance than the current generation of NAND-flash storage. We put one Optane device in our server with a baseline 4kB block-level random read latency of 11.14μs. In this setup, FlashNet delivers 28.14μs of remote 4kB page access latency. In comparison, the 4kB DRAM page access latency is 17.8μs. Hence, FlashNet's read path to the Optane drive adds minimum overheads to the overall remote page access performance. This experiment validates the suitability of the thin and fast data path of FlashNet even for the next generation of NVM devices.

Table 3. 4kB I/O Latencies in Mixed, Hybrid RNIC-FlashNet Configurations

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

In reads, the server is transmitting, for writes, the server is receiving.

7.2 Applications

We demonstrate the applicability of FlashNet using two applications: a Key-Value Store and a distributed sorting workload.

7.2.1 Key-Value Store. Key-Value stores is one of the most popular applications that is known to benefit from RDMA operations. Multiple stores have been proposed, e.g., Pilaf [67], HERD [48], FaRM [19], and so on, which explore the tradeoffs 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 [31] 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.

On 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 are copied at the value table, on the same index as the free entry. If the bucket is full, then we first try to displace one of the other entries in the same bucket, and if that is unsuccessful, then we push the entry into a victim bucket. On a `get`, the matching bucket is searched until the key is found; if not found, then 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 I/O amplification on random operations, the KV store has the following amortized behavior: A `put` operation will result in a read I/O 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 I/O. 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 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 [82]. We deployed Aerospike over the same storage as our KV store,

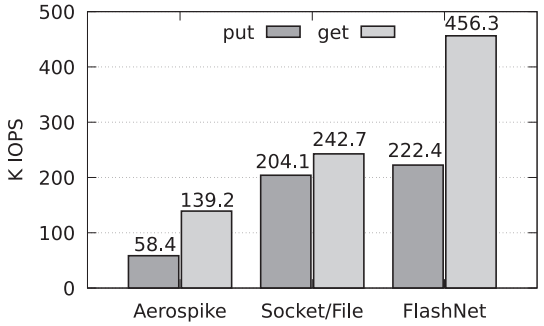

Fig. 10. Performance of 4kB puts and gets in the KV store.

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 4kB key-value size; we used 4kB to match the block size characteristics of the underlying storage and illustrate I/O 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.

7.2.2 Distributed Sorting on FlashNet. We have modified and ported RStore [90], 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 (of 15k) of code. The majority of these modifications are focused on acquiring RDMA-ready data buffers by `mmaping` ContigFS files at the server side. 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 that 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 four 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 I/O from the flash devices and does not incur any additional overheads. This observation can be verified by calculating the time difference and

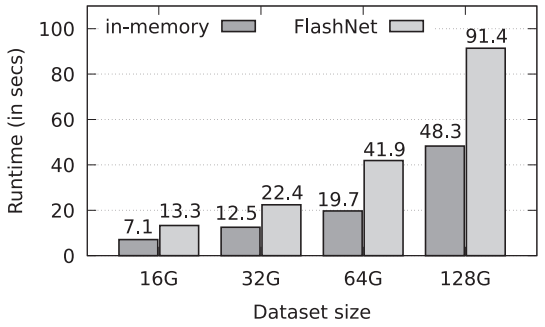

Fig. 11. Distributed sorting performance of RStore.

amount of data that are 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 43.1s, which is close to the expected NVMe device performances. With a simple calculation, we can verify that the read time should be 14.5s (128GB/(4 devices × 2.2GB/sec/device)) and the write time should be 35.5s (128GB/(4 devices × 0.9GB/sec/device)). That calculation gives us a total time of 50 seconds, whereas RStore/FlashNet adds 43.1s 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.

8 RELATED WORK

In this section, we cover the large body of related work in the field of high-performance network and storage integration. Table 4 summarizes our discussion here.

Inefficiencies in commodity end-host networking stacks have first been discussed in the 1990s and have led to the design of high-performance networking stacks [9, 10, 93, 98]. Modern day userspace RDMA network stacks such as InfiniBand, iWARP, or RoCE, are the latest incarnation of these ideas and are being used both in supercomputers as well as in data center workloads [19, 48, 67, 70, 83]. Meanwhile, commodity network and storage software stacks are constantly being optimized to eliminate inefficiencies. For example, networking efforts have focused on providing better locality [72], abstractions/APIs [29, 75], scalability [42], specialization [36, 64], per-packet processing [38, 60], and so on. In parallel, the storage community has also targeted offloading [11], direct hardware access [50], specialization [57, 78], fast I/O paths [81], scalability [8], interrupt coalescing [2], and polling [80, 101], and so on. Many of these efforts take a very *network-alike* approach towards reducing storage overheads [91]. FlashNet is built on such common high-performance I/O properties from both, networking and storage, and unifies them in a single stack. The easiest way to eliminate the data-path overheads when accessing storage is to use the `sendfile` interface. Nonetheless, this interface only solves the transmission-side issues and yet still requires the server application to be involved for cross-stack control loop transfers (data are not relayed through the userspace). Furthermore, it does not provide other highly desired client-side, high-performance I/O properties such as asynchronous I/O with selective notifications, direct polling of userspace mapped queues for low-latency, and so on.

In a systemwide approach, I/O-lite [71] unifies the data representation within a system for both the network and storage accesses. With a combination of read-only sharing and access control mechanisms, I/O-lite allows server applications, the file system, the cache/buffer, and the networking stack to concurrently share a single physical copy of the data. FlashNet's data page sharing

Table 4. Comparison of Related Work for Networked Storage Access in an End-to-End Manner

	Remote Flash Access	Server Involved?	Byte I/O	High-Perf. Properties	API	Comments
`sendfile`	Yes	Yes	Yes	No	file with socket	Only deals with the TX side issues
NASD [26]	Yes	No	Yes	No	distributed object store	Only deals with the server-side issues
SAN [12, 13]	Yes	No	No	No	block I/O	Benefits limited to the block layer
NAS [16, 59]	Yes	No	Yes	No	NFS, file I/O	Application must copy data to/from fs bufs
BGAS [23, 77]	No	N/A	Yes	Yes	RDMA queues	Limited to local flash access
Corfu [4]	Yes	No	Yes	No	distributed shared log	Uses custom flash access protocol
RDMA [19, 48, 70]	No	N/A	Maybe	Maybe	KV, distributed memory	Do not involve remote storage access
NVMeF [35, 69]	Yes	No	No	No	RDMA queues	*Block-level* remote flash access using RDMA
FlashNet	Yes	No	Yes	Yes	RDMA queues	*Byte-level* remote flash access using RDMA

Byte I/O means whether a client-side I/O request needs to rerform a read-modify-write cycle or the remote target accepts byte-level I/O requests.

mechanism between the flash controller, RDMA controller, and file system follows this principle. Many OSes designs such as Exokernel [21, 47], and the recently proposed Dataplane OSes (e.g., Arrakis [73] and IX [7]) focus on eliminating or reducing unnecessary kernel involvement by reducing its role to resource management only. However, their networking and storage stacks still run in isolation, without a cross-stack, end-to-end focus on performance for networked storage accesses.

Many file- and block-level distributed storage systems have been developed to deliver high-performance to applications. The Network-Attached Secure Disk (NASD) [26, 27] 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 [23, 77] uses RDMA operations but only to access *local* flash storage. Corfu [4] provides a distributed, shared log abstraction over a cluster of flash devices that can be directly attached to the network with the help of FPGA devices. The FPGA devices run the FTL and the network controllers. Unlike FlashNet, which supports a complete set of RDMA operations (not tied to any storage abstraction) on any flash device, Corfu requires SSD devices to implement specific operations, e.g., a seal, and write-once operations. Recent work in storage disaggregation has identified network and storage protocol processing related overheads to be a performance bottleneck [28, 51]. Naturally, there are previous works on improving performance of block-level [12, 43, 56, 66, 99] and file-level storage accesses [16, 32, 96] and even the integration of RDMA in them [13, 14, 54, 59]. Distributed file systems, such as DAFS [62, 63], DFS [17], and NFS [59], GlusterFS

[30], and so on, 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 [44].

In comparison to the aforementioned NAS/SAN approaches, FlashNet only provides a mechanism to unify storage and network processing concerns without imposing its usage. Similarly 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 I/O; 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 [19, 25, 48, 70] as well as JVM-based distributed systems [61, 84]. Distributed data platforms such as Apache Crail [37, 84] 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. RDMA has also been used as an optimized networking technology for distributed operations on persistent memory to build a shared-memory abstraction [79], replication [103], transactions [94], and so on. Their use of RDMA operation is limited to emulated persistent memory buffers in DRAM. With the use of FlashNet, these RDMA operations can be extended to flash/NVM locations as well. In general, applying RDMA to remote storage/persistent memory brings additional challenges, which are discussed at length by Hoefler et al. [33].

More recently, Project Donard [6] and NVMe over Fabrics (NVMeF) [35, 69] transport 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 richer RDMA semantics (one-sided operations, byte-addressability, asynchronous I/O, etc.) for remote storage access, whereas NVMe and hence NVMeF as well currently only support block-level device accesses. Hence, for unaligned requests, an application must emulate a read-modify-write cycle over the network. 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.

Looking beyond block-level accesses, there have been recent efforts to support byte-level local and remote NVM accesses. The Persistent Memory Programming (PMEM) project (www.pmem.io) aims to build new programing abstractions for byte-level, directly addressable NVM storage. These abstractions are used for the Direct Access (DAX) feature for multiple file systems including `ext4`. The project also includes support for remote persistent memory using RDMA operations. PMEM and FlashNet share the same common goal of providing byte-level remote persistent storage accesses, albeit, using different levels of abstractions. The use of FlashNet can enable PMEM to access the current-generation of block-level NVMe devices too.

Like FlashNet, ReFlex [52] takes a holistic approach towards optimizing network as well as storage I/O 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 applications 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 [58]. 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.

9 CONCLUSION

In this article, we have presented FlashNet, a unified software I/O 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 I/O hierarchy, and unifies efforts across the network and storage stacks.

ACKNOWLEDGMENTS

We thank Edward Bortnikov, Edouard Bugnion, Torsten Hoefler, and the anonymous reviewers for their helpful comments and feedback. Zoltan Nagy generously provided hardware loans for experiments in this article.

REFERENCES

[1] 2018. RDMA communication manager API. Retrieved July 2018 from https://linux.die.net/man/7/rdma_cm.

[2] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. 2011. vIC: Interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 USENIX Conference (USENIX ATC'11)*. USENIX Association, Berkeley, CA, 45–58.

[3] Jens Axboe. 2018. Flexible I/O tester. Retrieved July 2018 from https://linux.die.net/man/1/fio.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, 1–14.

[5] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (Mar. 2017), 48–54. DOI: https://doi.org/10.1145/3015146

[6] Stephen Bates. 2015. Donard: NVM Express for Peer-2-Peer between SSDs and other PCIe Devices. Retrieved July 2018 from http://www.snia.org/sites/default/files/SDC15_presentations/nvme_fab/StephenBates_Donard_NVM_Express_Peer-2_Peer.pdf.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. 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)*. USENIX Association, Berkeley, CA, 49–65.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, New York, NY, Article 22, 10 pages. DOI: https://doi.org/10.1145/2485732.2485740

[9] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. 1994. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. IEEE Computer Society Press, Los Alamitos, CA, 142–153. DOI: https://doi.org/10.1145/191995.192024

[10] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. 1996. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM, New York, NY, 245–259. DOI: https://doi.org/10.1145/238721.238784

[11] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, 387–400. DOI: https://doi.org/10.1145/2150976.2151017

[12] Adrian M. Caulfield and Steven Swanson. 2013. QuickSAN: A storage area network for fast, distributed, solid state disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 464–474. DOI: https://doi.org/10.1145/2485922.2485962

[13] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. 2003. 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)*. ACM, New York, NY, 209–219. DOI: https://doi.org/10.1145/944747.944754

[14] Lei Chai, Xiangyong Ouyang, Ranjit Noronha, and Dhabaleswar K. Panda. 2007. 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)*. ACM, New York, NY, 5–11. DOI: https://doi.org/10.1145/1374596.1374599

[15] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4 (Nov. 2004), 837–863. DOI: https://doi.org/10.1145/1027794.1027801

[16] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffrey Lefebvre, Daniel Ferstay, and Andrew Warfield. 2014. Strata: Scalable high-performance storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Berkeley, CA, 17–31.

[17] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. 2003. The direct access file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, Berkeley, CA, 175–188.

[18] Chet Douglas. 2015. RDMA with PMEM: Software mechanisms for enabling access to remote persistent memory. Retrieved July 2018 from http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, 401–414.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 54–70. DOI: https://doi.org/10.1145/2815400.2815425

[21] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, New York, NY, 251–266. DOI: https://doi.org/10.1145/224056.224076

[22] Roman Pletka et al. 2018. Management of next-generation NAND flash to achieve enterprise-level endurance and latency targets (unpublshed).

[23] Blake G. Fitch et al. 2013. Blue Gene Active Storage (BGAS) for High Performance BG/Q I/O and Scalable Data-centric Analytics. Retrieved July 2018 from https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/bgas/bgas-fitch.pdf?__blob=publicationFile.

[24] Philip Werner Frey. 2010. *Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks*. Ph.D. Dissertation. ETH Zurich. DOI: https://doi.org/10.3929/ethz-a-006133695 Dissertation Number 19001.

[25] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS'09)*. IEEE Computer Society, Washington, DC, 553–560. DOI: https://doi.org/10.1109/ICDCS.2009.32

[26] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. 1998. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, 92–103. DOI: https://doi.org/10.1145/291069.291029

[27] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. 1997. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*. ACM, New York, NY, 272–284. DOI: https://doi.org/10.1145/258612.258696

[28] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. 2017. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR'17)*. ACM, New York, NY, Article 16, 9 pages. DOI: https://doi.org/10.1145/3078468.3078483

[29] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. 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)*. USENIX Association, Berkeley, CA, 135–148.

[30] Red Hat. 2018. GlusterFS. Retrieved July 2018 from http://www.gluster.org/.

[31] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC'08)*. Springer-Verlag, Berlin, 350–364. DOI: https://doi.org/10.1007/978-3-540-87779-0_24

[32] Dean Hildebrand and Peter Honeyman. 2005. 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)*. IEEE Computer Society, Los Alamitos, CA, 18–27. DOI: https://doi.org/10.1109/MSST.2005.14

[33] Torsten Hoefler, Robert B. Ross, and Timothy Roscoe. 2015. Distributing the data plane for remote storage access. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HotOS'15)*. USENIX Association, Berkeley, CA.

[34] Xiao-Yu Hu, Robert Haas, and Eleftheriou Evangelos. 2011. Container marking: Combining data placement, garbage collection and wear levelling for flash. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'11)*. IEEE Computer Society, Los Alamitos, CA, 237–247. DOI: https://doi.org/10.1109/MASCOTS.2011.50

[35] NVM Express Inc. 2016. NVM Express over Fabrics Specification 1.0. Retrieved July 2018 from http://www.nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605-1.pdf.

[36] Solarflare Communications Inc. 2018. OpenOnload. Retrieved July 2018 from http://www.openonload.org/.

[37] Apache Crail (Incubating). 2018. A High-Performance Distributed Data Store for the Apache Ecosystem. Retrieved July 2018 from http://crail.incubator.apache.org/.

[38] Intel. 2018. DPDK: Data Plane Development Kit. Retrieved July 2018 from http://dpdk.org/.

[39] Intel. 2018. Intel Optane SSD 900P Series. Retrieved July 2018 from https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series.html.

[40] Intel. 2018. Intel's 3D XPoint Technology Products—What's Available and What's Coming Soon. Retrieved July 2018 from https://software.intel.com/en-us/articles/3d-xpoint-technology-products.

[41] Nikolas Ioannou, Kornilios Kourtis, and Ioannis Koltsidas. 2018. Elevating commodity storage with the SALSA host translation layer. In *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*. 277–292.

[42] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. 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)*. USENIX Association, Berkeley, CA, 489–502.

[43] Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry. 2005. A scalable and high performance software iSCSI implementation. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4 (FAST'05)*. USENIX Association, Berkeley, CA, 267–280.

[44] Scott M. Johnson. 2014. Violin and Microsoft's High-Performance, All-Flash Enterprise Storage. Retrieved July 2018 from https://insightsblog.violinsystems.com/blog/violin-and-microsoft-windows-flash-array.

[45] Rick Jones et al. 2018. Netperf: A network performance benchmark. Retrieved July 2018 from https://github.com/HewlettPackard/netperf.

[46] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA, 85–100.

[47] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 52–65. DOI: https://doi.org/10.1145/268998.266644

[48] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 295–306. DOI: https://doi.org/10.1145/2619239.2626299

[49] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. 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)*. USENIX Association, Berkeley, CA, 185–201.

[50] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*. USENIX Association, Berkeley, CA, 41–45. http://dl.acm.org/citation.cfm?id=3026852.3026861

[51] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 29, 15 pages. DOI: https://doi.org/10.1145/2901318.2901337

[52] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote flash == Local flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 345–359. DOI: https://doi.org/10.1145/3037697.3037732

[53] Kenneth C. Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. DOI: https://doi.org/10.1145/365628.365655

[54] Evangelos Koukis, Anastassios Nanos, and Nectarios Koziris. 2010. GMBlock: Optimizing data movement in a block-level storage sharing system over myrinet. *Cluster Comput.* 13, 4 (Dec. 2010), 349–372. DOI: https://doi.org/10.1007/s10586-009-0106-y

[55] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 273–286.

[56] Edward K. Lee and Chandramohan A. Thekkath. 1996. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, 84–92. DOI: https://doi.org/10.1145/237090.237157

[57] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. 2016. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, 339–353.

[58] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. 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)*. ACM, New York, NY, 449–466. DOI: https://doi.org/10.1145/3037697.3037710

[59] Bo Li, Panyong Zhang, Zhigang Huo, and Dan Meng. 2009. Early experiences with write-write design of NFS over RDMA. In *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage (NAS'09)*. IEEE Computer Society, Los Alamitos, CA, 303–308. DOI: https://doi.org/10.1109/NAS.2009.58

[60] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. 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)*. USENIX Association, Berkeley, CA, 429–444.

[61] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K. Panda. 2016. High-performance design of apache spark with RDMA and its benefits on various workloads. In *Proceedings of the IEEE International Conference on Big Data*. 253–262.

[62] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, and Margo I. Seltzer. 2003. Making the most out of direct-access network attached storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, Berkeley, CA, 189–202.

[63] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer, Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv Wickremesinghe, and Eran Gabber. 2002. Structure and performance of the direct access file system. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC'02)*. USENIX Association, Berkeley, CA, 1–14.

[64] Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 175–186. DOI: https://doi.org/10.1145/2619239.2626311

[65] Bernard Metzler. 2018. SoftiWARP: Software iWARP kernel driver and user library for Linux. Retrieved July 2018 from https://github.com/zrlio/softiwarp.

[66] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. 2014. 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)*. USENIX Association, Berkeley, CA, 257–273. http://dl.acm.org/citation.cfm?id=2616448.2616473.

[67] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. 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)*. USENIX Association, Berkeley, CA, 103–114.

[68] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-volatile storage. *Commun. ACM* 59, 1 (Dec. 2015), 56–63. DOI: https://doi.org/10.1145/2814342

[69] Wael Noureddine. 2015. Implementing NVMe over Fabrics. Retrieved July 2018 from http://www.snia.org/sites/default/files/SDC15_presentations/networking/WaelNoureddine_Implementing_%20NVMe_revision.pdf.

[70] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud storage system. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. DOI: https://doi.org/10.1145/2806887

[71] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. IO-lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX Association, Berkeley, CA, 15–28.

[72] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 337–350. DOI: https://doi.org/10.1145/2168836.2168870

[73] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 1–16. http://dl.acm.org/citation.cfm?id=2685048.2685050

[74] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. 2015. A hybrid I/O virtualization framework for RDMA-capable network interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*. ACM, New York, NY, 17–30. DOI: https://doi.org/10.1145/2731186.2731200

[75] Luigi Rizzo. 2012. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, 101–112.

[76] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comp. Syst.* 10, 1 (Feb. 1992), 26–52. DOI: https://doi.org/10.1145/146941.146943

[77] Felix Schürmann, Fabien Delalondre, Pramod S. Kumbhar, John Biddiscombe, Miguel Gila, Davide Tacchella, Alessandro Curioni, Bernard Metzler, Peter Morjan, Joachim Fenkes, Michele M. Franceschini, Robert S. Germain, Lars Schneidenbach, T. J. Ward, and Blake G. Fitch. 2014. Rebasing I/O for scientific computing: Leveraging storage class memory in an IBM bluegene/Q supercomputer. In *Proceedings of the 29th International Conference on Supercomputing (ISC'14)*, Vol. 8488. Springer-Verlag, New York, 331–347. DOI: https://doi.org/10.1007/978-3-319-07518-1_21

[78] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 67–80.

[79] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 323–337. DOI: https://doi.org/10.1145/3127479.3128610

[80] Dong In Shin, Young Jin Yu, Hyeong S. Kim, Jae Woo Choi, Do Yung Jung, and Heon Y. Yeom. 2013. 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)*. USENIX Association, Berkeley, CA.

[81] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. 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)*. USENIX Association, Berkeley, CA, 483–488.

[82] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a real-time operational DBMS. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1389–1400. DOI: https://doi.org/10.14778/3007263.3007276

[83] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. 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)*. USENIX Association, Berkeley, CA, 347–353.

[84] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Bull. Techn. Committee on Data Eng.* 40, 1 (Mar. 2017), 40–52.

[85] Nisha Talagala. 2012. Native Flash Support for Applications. Retrieved July 2018 from http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120823_S304B_Talagala.pdf.

[86] Tom Talpey. 2015. Remote Access to Ultra-low Latency Storage. Retrieved July 2018 from https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf.

[87] Mellanox Technologies. 2018. RDMA Aware Networks Programming User Manual. Retrieved July 2018 from http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[88] Mellanox Technologies. 2018. Software RDMA over Converged Ethernet (RoCE). Retrieved July 2018 from https://github.com/SoftRoCE.

[89] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. 2011. A case for RDMA in clouds: Turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11)*. ACM, New York, NY, Article 17, 5 pages. DOI: https://doi.org/10.1145/2103799.2103820

[90] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R. Gross. 2015. RStore: A direct-access DRAM-based data store. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*. 674–685.

[91] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, and Thomas R. Gross. 2013. Unified high-performance I/O: One stack to rule them all. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA.

[92] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 306–324. DOI: https://doi.org/10.1145/3132747.3132762

[93] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the F15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, New York, NY, 40–53. DOI: https://doi.org/10.1145/224056.224061

[94] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 87–104. DOI: https://doi.org/10.1145/2815400.2815419

[95] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 111–118.

[96] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, Article 2, 17 pages.

[97] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[98] John Wilkes. 1992. *Hamlyn—An Interface for Sender-based Communications*. Technical Report HPL-OSR-92-13. Palo Alto, CA.

[99] Dimitrios Xinidis, Angelos Bilas, and Michail D. Flouris. 2005. 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)*. IEEE Computer Society, Los Alamitos, CA, 261–269. DOI: https://doi.org/10.1109/MSST.2005.23

[100] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. 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)*. ACM, New York, NY, Article 6, 11 pages. DOI: https://doi.org/10.1145/2757667.2757684

[101] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, 25–32.

[102] Yiwen Zhang, Juncheng Gu, Youngmoon Lee, Mosharaf Chowdhury, and Kang G. Shin. 2017. Performance isolation anomalies in RDMA. In *Proceedings of the Workshop on Kernel-Bypass Networks (KBNets'17)*. ACM, New York, NY, 43–48. DOI: https://doi.org/10.1145/3098583.3098591

[103] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 3–18. DOI: https://doi.org/10.1145/2694344.2694370

Received April 2018; accepted July 2018