Flash Storage Disaggregation

Ana Klimovic
Stanford University
anakli@stanford.edu

Christos Kozyrakis
Stanford University
kozyraki@stanford.edu

Eno Thereksa
Confluent Inc. and
Imperial College London
eno@confluent.io

Binu John
Facebook Inc.
binu@fb.com

Sanjeev Kumar
Facebook Inc.
skumar@fb.com

Abstract

PCIe-based Flash is commonly deployed to provide datacenter applications with high IO rates. However, its capacity and bandwidth are often underutilized as it is difficult to design servers with the right balance of CPU, memory and Flash resources over time and for multiple applications. This work examines Flash disaggregation as a way to deal with Flash overprovisioning. We tune remote access to Flash over commodity networks and analyze its impact on workloads sampled from real datacenter applications. We show that, while remote Flash access introduces a 20% throughput drop at the application level, disaggregation allows us to make up for these overheads through resource-efficient scale-out. Hence, we show that Flash disaggregation allows scaling CPU and Flash resources independently in a cost effective manner. We use our analysis to draw conclusions about data and control plane issues in remote storage.

Categories and Subject Descriptors H.3.4 [*Systems and Software*]: Performance Evaluation

General Terms Performance, Measurement

Keywords Network storage, Flash, Datacenter

1. Introduction

Flash is increasingly popular in datacenters of all scales as it provides high throughput, low latency, non-volatile storage. Specifically, PCIe-based Flash devices offer 100,000s of IO operations per second (IOPS) and latencies in the 10s of μs range [17, 20]. Such devices are commonly used to support persistent, key-value stores (KVS) with high throughput requirements. At Facebook, for example, many applications that generate web-page content use PCIe Flash. Similarly, LinkedIn reports using PCIe SSDs to scale its distributed key-value database, Project Voldemort [45], to process over 120 billion relationships per day [28].

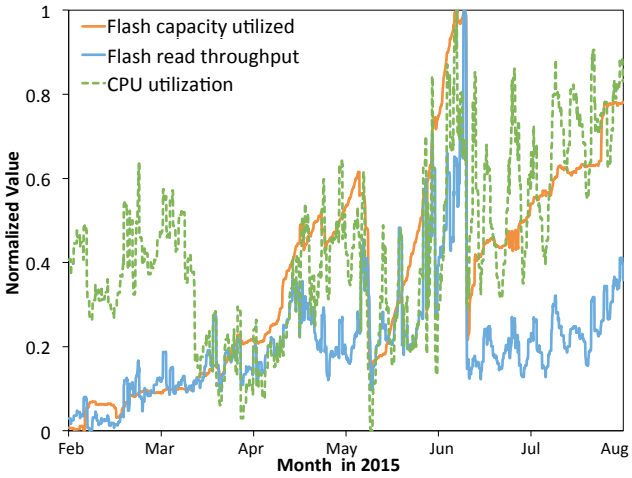


Figure 1: Sample resource utilization on servers hosting a Flash-based key-value store service at Facebook, normalized over a 6 month period. Flash and CPU utilization vary over time and scale according to separate trends.

Designing server machines with the right balance of CPU, memory, and Flash is difficult because each application has unique and often dynamically varying requirements for each resource [9]. Figure 1 shows the Flash capacity, Flash read throughput, and CPU utilization on a set of servers hosting a real Facebook application that uses Flash for its key-value store. Each metric is normalized to its maximum value over a 6 month period. Flash capacity, Flash throughput, and CPU utilization vary over time and follow separate trends. For example, mid April to May, Flash capacity utilization increases while Flash read throughput decreases and CPU utilization oscillates. Resource utilization studies of the Microsoft Azure cloud computing platform and a large

EuroSys '16 April 18–21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901337

private cloud at Google have also shown that storage capacity and IO rates are not necessarily correlated [46, 68].

The lack of balance leads to deploying machines with significantly overprovisioned resources, which can increase the total cost of ownership [25]. In Figure 1, Flash capacity, IOPS, and compute resources are under-utilized for long periods of time. Overprovisioning IOPS is particularly common when deploying PCIe Flash in datacenters because various software overheads, referred to as "datacenter tax" by Kanev et al. [39], often cause storage tier services to saturate CPU cores before saturating PCIe Flash IOPS. In general, the performance of PCIe-based Flash is so high that utilizing the resource effectively is a challenge [15]. Baidu reported their storage system's realized IO bandwidth was only 50% of their Flash hardware's raw bandwidth [56]. Similarly, Flash capacity is often underutilized as servers are deployed with enough Flash to satisfy projected future demands. Deploying high-capacity Flash devices provides on-demand flexibility as application requirements vary or spike. High-capacity devices also better amortize the cost of the Flash controller.

Resource disaggregation has been proposed to deal with the challenge of imbalanced resource requirements and the resulting overprovisioning of datacenter resources [26]. By physically decoupling resources, datacenter operators can more easily customize their infrastructure to maximize the performance-per-dollar for target workloads. This approach is already common for disk storage [18]. Network overheads are small compared to a disk's millisecond access latency and low 100s of IOPS, so it is common for applications to remotely access disks deployed in other machines.

In this paper, we analyze a similar approach of disaggregating Flash from CPU and memory in servers hosting datastore applications. We use the term "disaggregated Flash" (or remote Flash) to refer to Flash that is accessed over a high-bandwidth network, as opposed to Flash accessed locally over a PCIe link. Remote Flash accesses can be served by a high-capacity Flash array on a machine dedicated to serving storage requests over the network. Alternatively, we can enable remote access to Flash on a nearby server, which itself runs Flash-based applications, but has spare Flash capacity and IOPS that can be shared over the network. With either approach, disaggregation can span racks, rows, or the whole datacenter.

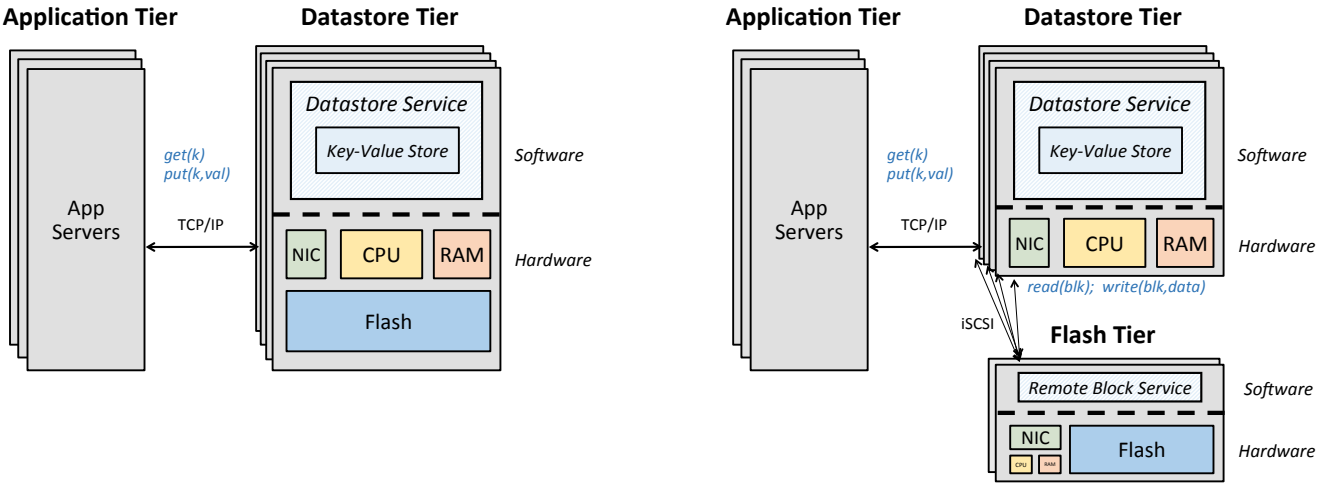
We show that disaggregating Flash can lead to significant resource utilization benefits (as high as 40%) by allowing Flash resources to scale independently from compute resources. However, remote access to Flash also introduces performance overheads and requires deploying extra resources for network protocol processing. With this performance-cost trade-off in mind, we answer the following questions:

1. **Can datacenter applications tolerate the performance overhead of remote access to Flash with existing network storage protocols?** We show that key-value store workloads sampled from real applications at Facebook have acceptable end-to-end performance when accessing remote Flash using the iSCSI network storage protocol (Section 4.2). Remote versus local access to Flash increases tail latency by 260μs, while our target applications have latency SLAs on the order of several milliseconds. Protocol processing overhead reduces the peak number of queries processed per second by approximately 20% on average. Although this throughput degradation is not negligible, we show that disaggregation allows us to compensate for this throughput loss by independently and efficiently scaling CPU and Flash resources.

2. **How can we optimize the performance of a remote Flash server?** We find the following optimizations particularly useful for tuning the performance of a remote Flash server: parallelizing protocol processing across multiple threads, enabling jumbo frames and packet processing offloads on the NIC, spreading interrupt affinity across CPU cores and pinning protocol processes that share TCP connection state to the same core. With these optimizations a remote Flash server shared between 6 IO-intensive tenants can support over 1.5$\times$ more IOPS than with an out-of-the-box setup (Section 3.3). For our target applications, these optimizations improve end-to-end throughput by 24% (Section 4.2).

3. **When does disaggregating Flash lead to significant resource utilization benefits?** We compare server resource requirements for hosting applications with direct-attached Flash and disaggregated Flash, showing that disaggregation can lead to 40% resource savings at the same throughput level (Section 4.3). Disaggregating Flash is most beneficial when the ratio of compute to storage requirements varies over time and/or differs widely between applications.

We also discuss the implications of our study for future work on remote Flash access (Section 5). We show that for end-to-end application performance, the majority of iSCSI overhead is masked by CPU overhead *above* the network storage layer, which arises from computational intensity in the datastore application and the generic RPC frameworks used for inter-tier communication. We motivate the need for a lower overhead dataplane for remote Flash access, particularly for applications that issue more than 10,000s of IOPS or have sub-millisecond latency SLAs. Moreover, we briefly discuss implications for control plane resource management at the cluster and storage node level.



(a) Direct-attached (local) Flash

(b) Disaggregated (remote) Flash

Figure 2: Target application architecture. Datastore tier servers host key-value store services on local or remote Flash.

2. Background

2.1 Flash-Backed, Key-Value Stores

Terabyte and even petabyte-scale key-value store (KVS) services have become components of large-scale web applications. Examples of persistent key-value stores embedded in datacenter applications include RocksDB, LevelDB, and Project Voldemort [19, 21, 45]. These Flash-backed datastores commonly store various website user state such as viewing history.

Tiered architecture: Our study targets datacenter applications with a multi-tier architecture shown in Figure 2a. Applications divide datasets into replicated partitions, spread across multiple servers in the datastore tier. The application tier issues requests to the datastore tier and may also interface with a front-end web tier (not shown in the figure). Servers in the datastore tier host services with an embedded key-value store and manage access to Flash storage. In Figure 2a, the storage is local, direct-attached Flash. Our study looks at the implications of the architecture shown in Figure 2b, where datastore servers access Flash over the network on a remote server in the Flash tier.

Compute intensity: Datastore services are often compute-intensive, particularly when handling complex queries, compactions or data integrity checks. Serialization and deserialization further increase CPU utilization on datastore servers as they communicate with application tier clients using generic frameworks like Thrift or Protocol Buffers [5, 23]. The CPU overhead associated with inter-tier communication, part of the "datacenter tax" characterized by Kanev et. al [39], limits IO rates on datastore servers to only a fraction of the capabilities of today's PCIe Flash devices.

Use of Flash: Although our target applications do not saturate the 100,000s of IOPS that PCIe Flash offers, the workloads still issue high rates of small random reads which could not be supported by disk (e.g., 10,000s IOPS of 4 kB random reads). Other characteristics of the applications include high write throughput (e.g., 500 MB/s of sequential writes) and round-trip latency SLAs on the order of 5 to 10 ms. Maximizing IOPS is more important than reducing latency for these applications. PCIe Flash provides high IOPS by using multiple channels to communicate with the underlying NAND storage [50]. Multiple channels also help eliminate the latency associated with requests queuing, another important optimization.

RocksDB: Our study uses the RocksDB database [19], since it is an example of a KVS deployed at Facebook and embedded in the real applications we model. RocksDB is not a distributed service, but rather a high-performance single-node engine that stores keys and values (arbitrary byte arrays) in a log-structured merge (LSM) tree. It builds on LevelDB, scales to multiple cores, uses Flash efficiently, and exposes parameters for application-specific performance tuning. RocksDB does not provide failover or consistency guarantees, but applications can build these features on top of RocksDB, if required. Our target applications have weak consistency requirements.

2.2 Remote Storage Protocols

There are many existing protocols and mechanisms that enable remote access to data. We outline protocol-level requirements for remote access to Flash in the datacenter and discuss how existing protocols align with these requirements, leading to our choice of protocol for this study.

Requirements: First, we require a protocol that provides applications with the illusion of a local Flash device, abstracting remote access and requiring *minimal changes to application code*. Second, the protocol should *run on commodity networking hardware*. Protocols that leverage the existing Ethernet infrastructure in datacenters greatly simplify

and reduce the cost of deploying disaggregated Flash. Third, the protocol should *scale to a datacenter scale*, enabling datastore servers to access remote Flash across racks and even clusters. Fourth, the protocol should have *low performance overhead* for our applications, supporting high IOPS and low latency storage access.

Network file sharing protocols: Network File System (NFS) [60] and Server Message Block (SMB) are examples of protocols used in network attached storage (NAS) architectures. They provide a file abstraction to storage and manage consistency. NFS and SMB run on commodity Ethernet but have limited scalability since clients access storage through dedicated network file servers. Distributed file systems like HDFS [66] and the Google File System [22] scale well but transfer data in large, megabyte-size chunks which have high overhead for small key-value data accesses.

Block IO protocols: Block IO protocols are commonly used in storage attached network (SAN) systems to give clients the illusion of local block storage. Internet Small Computer System Interface (iSCSI) is a widely used protocol that encapsulates SCSI commands into TCP packets, allowing datacenter-wide communication over commodity Ethernet networks. Fibre Channel Protocol (FCP) also transports SCSI commands but requires a dedicated, costly Fibre Channel network. ATA over Ethernet (AoE) is a less mature protocol that transmits block IO requests over raw Ethernet. Since the protocol does not run on top of IP, AoE avoids TCP/IP overheads but the protocol is not routable, limiting disaggregation to within a rack.

RDMA-based protocols: Remote Direct Memory Access (RDMA) is a remote memory management capability that allows server-to-server data movement between application memory without involving the CPU. By offloading remote access to the network card, RDMA eliminates the need to copy data between user and kernel buffers. First deployed in high-performance computers, RDMA has since expanded into Ethernet networks [49]. However, RDMA over Converged Ethernet still requires specialized (RDMA-capable) NIC hardware and the protocol assumes a lossless fabric, limiting scalability. RDMA has been used for initial implementations of NVMe over Fabrics [12]. Non-Volatile Memory Express (NVMe) is an optimized interface for high-performance storage and the goal of NVMe over Fabrics is the provide efficient access to local and remote stroage with a transport-neutral specification [55].

PCIe-based Fabrics: PCI Express (PCIe) was originally developed as a high-bandwidth point-to-point interconnect between CPUs and peripherals. Recent PCIe switch designs enable a unified PCIe backplane interconnect or "fabric" that enables host-to-host communication and IO resource sharing [6, 69]. Although high performance, PCIe remains expensive and its complex tree architecture limits scalability.

For our study, we choose to use the iSCSI protocol[1] since it is a widely deployed protocol that satisfies the first three requirements outlined above: it provides a block IO abstraction, runs on commodity Ethernet and supports datacenter-wide communication. The fourth requirement—performance—is a potential concern with iSCSI since the protocol involves substantial processing and was originally designed for remote disk, not Flash [37, 61]. In Section 3.3, we show how we tune the performance of a remote Flash server to keep iSCSI overhead low enough for our target applications. The methodology we present in this paper for evaluating the implications of remote Flash is applicable to other storage protocols. By using a higher overhead protocol in our study, we ensure our conclusions are conservative. We performed some measurements with the SMB protocol and found the performance overheads to be similar to iSCSI.

2.3 Related Work

Disaggregated storage: Disaggregation is a well-known approach for independently scaling resources [26]. Disaggregating disk storage from compute nodes is common because network access does not introduce noticeable overhead for disk, which is slow and low-throughput to begin with [2]. Thus, in cloud and enterprise environments, block storage is commonly virtualized and backed by remote storage which can be adjusted to meet application requirements [1, 72]. Systems such as Petal [40], Parallax [75], and Blizzard [51] implement distributed virtual block stores for disk storage to abstract the storage layer from clients while providing good performance and features such as replication and failure recovery.

Disaggregating high-performance Flash is more challenging since the network imposes a larger percentage overhead. Our study differs from previous work on remote storage by focusing on quantifying the remote access overhead for high-performance PCIe Flash and understanding its end-to-end impact for real datacenter applications. Contrary to systems such as CORFU [7] and FAWN [3], which propose using remote Flash arrays as a distributed shared log, we analyze performance for remote Flash exposed as a traditional block device. The block interface is compatible with a wide variety of applications and offers greater flexibility for remote Flash use-cases compared to approaches that impose distributed shared log semantics.

A lot of recent work on disaggregation focuses on the rack scale [8, 14, 18, 30, 33, 63] whereas in our study, disaggregation can span the datacenter.

Lim et al. [43, 44] apply disaggregation higher up the memory hierarchy. They observe that memory footprints of enterprise workloads vary across applications and over time and show that adding disaggregated memory as second-tier

[1] Although we evaluate the performance of our architecture for workloads modeled based on Facebook applications, our choice of protocol for this study is independent and does not reflect Facebook's choice of protocol for disaggregated Flash in its datacenters.

capacity can improve application performance. We similarly observe that Flash utilization varies across datacenter workloads and over time. Our study of disaggregated Flash aims to analyze the tradeoff between performance and resource utilization.

Remote storage protocols: There is previous work on iSCSI performance tuning [37, 47, 76] and comparisons to other storage protocols [59] in the context of enterprise workloads. Offloading fixed-cost functionality to hardware using RDMA is an emerging theme [11, 12, 38, 49]. Our approach to optimizing remote Flash access is orthogonal and focuses on tuning system settings such as interrupt and process affinity, similar to techniques Leverich and Kozyrakis use to reduce interference of latency critical workloads [42].

Workload-driven modeling: Other studies have analyzed the performance and cost implications of datacenter storage architectures. For example, Uysal et al. analyzed replacing disks with MEMS [70], Narayanan et al. analyzed replacing disks with SSDs [53]. To our knowledge, ours is the first study to specifically examine this trade-off for high-performance Flash hosting key-value store workloads sampled from real datacenter applications.

3. Disaggregated Flash Architecture

We describe the disaggregated Flash architecture and protocol we evaluate in our study.

3.1 Architecture Overview

Figure 2a shows a common architecture used to host Flash-backed, key-value services described in Section 2.1. The deployment assumes that each datastore server "owns" a Flash device, meaning applications hosted on the server have exclusive access to the direct-attached Flash. We want to change this constraint with disaggregation. By "disaggregating Flash", we mean enabling remote access to Flash over a high bandwidth network, with the goal of improving Flash utilization. This involves exposing Flash, with spare capacity and IOPS, to applications hosted on servers which may or may not be in the same rack.

We depict disaggregated Flash in Figure 2b, where Flash is physically decoupled from the datastore tier. In this architecture, we have two types of servers: 1) *datastore servers* in the datastore tier, which have powerful CPU cores and memory used for hosting the datastore service, and 2) *Flash storage servers* in the Flash tier, which have high-capacity PCIe Flash arrays and high-bandwidth network ports along with a limited amount of CPU cores and memory for network processing in the remote block service layer. The two types of servers can be deployed independently across the datacenter to meet application requirements. Each application is given as many servers of each kind as it needs. Each Flash server can serve as many applications as its capacity and IOPS capabilities allow. A Flash server can be a machine used for other purposes, as long as it has some spare Flash capacity and IOPS to share over the network and some spare CPU for protocol processing. Alternatively, a Flash server can be a high capacity server with multiple Flash devices, dedicated to serving over-the-network storage requests. This approach amortizes the cost of Flash controllers, decreasing the cost per gigabyte. As applications demand more capacity or IOPS, we can incrementally deploy Flash servers on demand, benefiting from new storage technology as it becomes available and lower prices as the technology matures, without waiting for similar advancements in CPU and memory technology.

Allocating capacity and IOPS on disaggregated Flash requires a coordination manager. The manager is responsible for binpacking applications into the available resources across the datacenter, under numerous constrains such as fault-tolerance. We focus on the dataplane for remote Flash, since achieving sufficiently high performance in the dataplane is a prerequisite. We discuss implications and future work for the control plane in Section 5.2.

3.2 Remote Flash Access with iSCSI

The iSCSI protocol introduces several layers of software overhead for IO operations. Applications issue `read` and `write` system calls, which go through the kernel block device layer to the SCSI subsystem, as if accessing a local block device. The iSCSI protocol encapsulates SCSI storage commands into iSCSI Protocol Data Units (PDUs) which are transmitted over a TCP connection. The client (often called an initiator) establishes a long-lived session with the remote storage server (called the target) [61]. We describe protocol processing in more detail below.

Write processing: The iSCSI initiator determines when to transmit data based on flow-control messages from the target. The initiator maintains a pointer to the SCSI buffer with data for the next PDU. The iSCSI layer constructs an iSCSI header and a gather list describing the header and payload of the iSCSI PDU. The kernel's TCP/IP stack then processes and sends the PDU to the target, which may involve copying data between SCSI and NIC buffers, depending on the implementation of the TCP/IP stack.

Read processing: The initiator receives packets from the target and uses direct memory access (DMA) to transfer packets from the NIC to kernel memory. The kernel processes network interrupts, performs TCP/IP processing to strip off Ethernet, IP and TCP headers, and enqueues the iSCSI PDU payload in a socket descriptor structure. Based on the payload headers, the iSCSI layer creates a scatter-list which the socket layer uses to copy iSCSI PDU data from TCP segments into a SCSI buffer. A final copy occurs from the SCSI buffer to a user-space buffer where the application can access the data.

Similar processing occurs at the target in the TCP/IP, iSCSI and SCSI layers. The target also translates SCSI commands to NVMe commands [55], a task the Linux NVMe driver implements for compatibility.

3.3 Remote Flash Performance Tuning

As described in Section 3.2, iSCSI introduces significant overhead as each Flash request traverses multiple layers of the operating system on both the tenant (initiator) and the remote server (target). Protocol processing adds latency to each request and, more importantly, can saturate CPU resources, leading to queuing and throughput degradation. Hence, it is necessary to optimize the system to achieve high IOPS for remote Flash accesses.

Setup: For the purposes of this study, our experimental setup consists of a dozen 2-socket Xeon E5-2630 @ 2.30GHz servers with 6 cores (12 hyperthreads) on each socket. Each server has 64 GB of DRAM and a SolarFlare SFC9020 10 GbE network card. For application tier servers we use both sockets, while for the datastore servers and remote Flash servers we use 1 socket. The servers are connected with a 64 $\times$ 10 GbE Arista switch. Our Flash devices are Intel P3600 PCIe cards with 400 GB capacity [34]. All servers run Ubuntu LTS 14.0.3 distribution, with a 3.13 Linux kernel and 0.8 version NVMe driver. We use the `iscsitarget` Linux software package to run the iSCSI block service on a remote Flash server. We use the `open-iscsi` package as the iSCSI initiator stack, which we run on datastore servers (more generally referred to as tenants in this section). To expose Flash to multiple tenants, we statically partition the storage device and run a separate iSCSI target service for each partition.

For our tuning experiments, tenants use `FIO` [35] to generate an IO-intensive 4 kB random read workload. With a single tenant, our baseline iSCSI Flash server supports 10.5K IOPS. The bottleneck is CPU utilization on the Flash server for network processing. When we run the same `FIO` workload locally on the Flash server, we saturate the Flash device at 250K IOPS. We tune several system knobs to improve the iSCSI Flash server's throughput. Figure 3 shows the performance benefit of each optimization applied to a Flash server serving 1, 3, and 6 tenants.

Multi-processing: The bottleneck in the single-tenant baseline test is CPU utilization on *one* of the Flash server's 6 cores. The iSCSI service at the remote Flash server consists of 1 `istd` process per iSCSI session (each tenant establishes 1 session) and a configurable number of `istiod` processes per session which issue IO operations to the device based on iSCSI commands. Using multiple `istiod` processes to parallelize IO processing improves throughput to 46.5K IOPS for a single tenant (4.5$\times$ improvement). The default `iscsitarget` setting invokes 8 `istiod` processes. We empirically find that using 6 processes leads to 10% higher performance. However, parallelism is still limited by the `istd` process's use of a single TCP connection per session.[2] When multiple tenants access the Flash server, processing is spread over multiple sessions (thus, multiple TCP connections). With 6 tenants and 6 `istiod` processes per tenant, we achieve 2.3$\times$ IOPS improvement compared to using 1 `istiod` process per tenant. With 8 tenants, we saturate the PCIe device's 250K IOPS. In addition to improving throughput, multi-processing reduces queuing latency by making the Flash server a multi-server queuing system. Hyper-threading does not improve performance so we disable it for our tests.

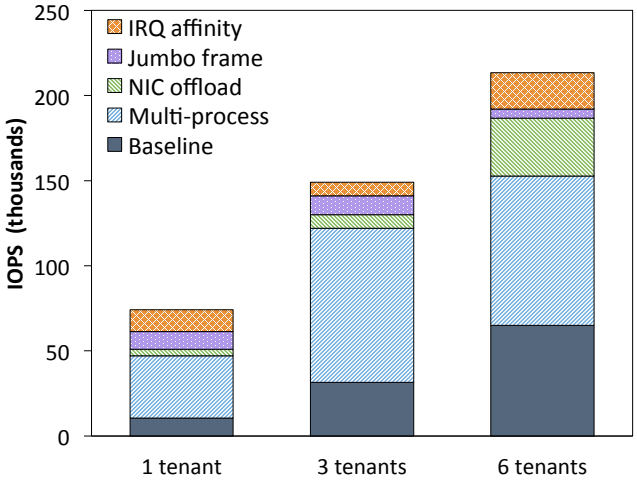


Figure 3: Datastore servers issue 4 kB random reads to a remote Flash server with 6 cores and a Flash device that supports up to 250K IOPS. We apply several optimizations to maximize IOPS.

NIC offloads: Enabling Transmit Segmentation Offload (TSO) and Large Receive Offload (LRO) on the NIC allows TCP segmentation to be processed in hardware, reducing CPU overhead and interrupts associated with dividing (assembling) 4 kB messages into (from) 1500-byte Ethernet frames. We find that TSO/LRO offloads increase aggregate IOPS on the remote Flash server by 8% with a single tenant (single TCP connection). The benefits of TSO/LRO increase with the number of tenants since multiple TCP connections benefit from the offload. In addition, the more loaded the server, the more important it is to reduce CPU utilization with NIC offloads. With 6 tenants, TSO/LRO provides a 22% increase in IOPS on the Flash server. For this read-only workload, we find it most beneficial to enable TSO only on the server (which sends data) and LRO only on the tenant (which receives data).

Jumbo frames: Enabling jumbo frames allows Ethernet frames up to 9000 bytes in length to be transmitted instead of the standard Ethernet maximum transfer unit (MTU) size of 1500 bytes. Although segmentation offloads on the NIC already eliminate most of the CPU overheads associated with splitting 4 kB blocks into multiple frames, jumbo frames eliminate the need for segmentation altogether and reduce the TCP header byte overhead, further increasing IOPS at the Flash server by 3 to 12%.

Interrupt affinity: We find it beneficial to distribute network traffic to separate queues on the NIC by hashing packet headers in hardware using Receive Side Scaling (RSS) [52].

[2] There exist proprietary versions of iSCSI that are multi-path, but the open-source iSCSI package in Linux is limited to 1 TCP connection per session.

We disable the `irqbalance` service in Linux and manually set the interrupt affinity for NIC and NVMe queues. To tune interrupt affinity, we consider the number of CPU cores available on the Flash server per tenant. We assume the number of tenants per Flash server is known (it is determined by the control plane). We describe how we manage interrupts on our 6-core Flash server in the case of 1, 3 and 6 tenants. With a single tenant, spreading NIC and NVMe Flash interrupts across all cores on the Flash server improves parallelism and leads to 21% more IOPS. With 3 tenants, for fairness, we allocate 2 server cores per tenant. To optimize performance in this scenario, we use `taskset` in Linux to pin the `istd` process associated with a tenant onto one core. We steer interrupts for the tenant's network queue to this core by manually setting rules in the NIC's flow table[3]. On the second core, we pin the session's `istiod` processes and steer NVMe Flash interrupts, achieving a 6% increase in IOPS. With 6 tenants, we only have 1 core available per tenant so we pin each session's `istd` and `istiod` processes on the same core and steer both NIC and NVMe interrupts for the corresponding tenant's traffic to that core. This helps minimize interference and improves IOPS by 11%.

In summary, we find that tuning system and network settings on a remote Flash server leads to substantial performance improvements. For 6 tenants, our optimized Flash server achieves 1.5× more IOPS than an out-of-the-box iSCSI setup (which uses the default 8 `istiod` processes per session). Our guidelines for tuning settings like interrupt affinity depend on the number of cores available per tenant on a remote Flash server. For IO-intensive workloads, iSCSI saturates 1 server core per tenant and benefits from additional cores for parallelizing interrupt management.

4. Evaluation

Given a tuned iSCSI setup for remote Flash access, we now evaluate the end-to-end impact of remote Flash on application performance and quantify the resource utilization benefits of disaggregation in various scenarios.

4.1 Experimental Methodology

Cluster: We use the same set of Xeon E5-2630 servers described in Section 3.3 to implement the architecture shown in Figure 2. Application tier clients generate `get` and `put` requests to the datastore tier which is made up of one or more datastore servers running a service with the embedded RocksDB key-value store [19]. The Flash tier in our experiments consists of a single server with a 400GB Intel P3600 PCIe Flash card and the 10GbE network infrastructure described in Section 3.3. Since RocksDB is a C++ library designed to be embedded directly into application code, we run a simple application called SSDB [31] as our datastore service to interface with RocksDB. SSDB is an event-based server wrapper for key-value stores like RocksDB which listens on a socket for requests coming from application tier clients and serializes and deserializes requests according to a text-based protocol.

To generate client load on the RocksDB datastore servers, we use the `mutilate` load generator [41]. The `mutilate` tool coordinates a large number of client threads across multiple machines to generate the desired QPS load, while a separate unloaded client measures latency by issuing one request at the time. By setting `get:put` ratios, key and value sizes, and key distributions in `mutilate` and also controlling the SSDB application's CPU intensity and memory utilization, we configure our setup to create workloads similar to real applications built on top of RocksDB at Facebook.

Workloads: Table 1 summarizes the IO patterns of four Facebook workloads which use the RocksDB key-value store. These applications typically issue 2,000 to 10,000 read operations to Flash per second per TB. Reads are random access and generally under 50 kB in size. Write operations occur when RocksDB flushes a memtable or performs a database compaction. Writes are sequential and significantly larger in size (up to 2MB) but less frequent than reads. Datastore servers hosting these applications with direct-attached PCIe Flash saturate CPU cores before saturating Flash IOPS due to the computational intensity of application code and inter-tier communication. Since the IO characteristics of the workloads in Table 1 do not differ significantly, we present results for Workload A and sweep various parameters like CPU intensity and the `get:put` ratio to better understand performance sensitivity to workload characteristics.

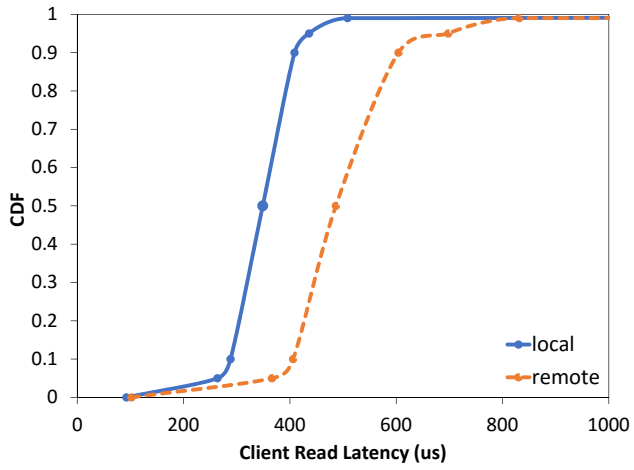
For our workloads, RocksDB keys are short strings (17 to 21 bytes), which can be used to store information such as user IDs. Values are longer strings which store various user state such as vieweing history. Data is continuously appended to values, up to a maximum length per value. In the case of Workload A, 100 byte user state is appended up to a total maximum of 10 kB. `get` requests processed by the SSDB application are issued to RocksDB reader threads, resulting in random read IOs up to 10 kB in size. `set` requests are interpreted as "append" operations by the SSDB application which uses RocksDB's `merge` operator to issue requests to writer threads. For our experiments, the SSDB application

Table 1: Facebook Flash Workload Characteristics

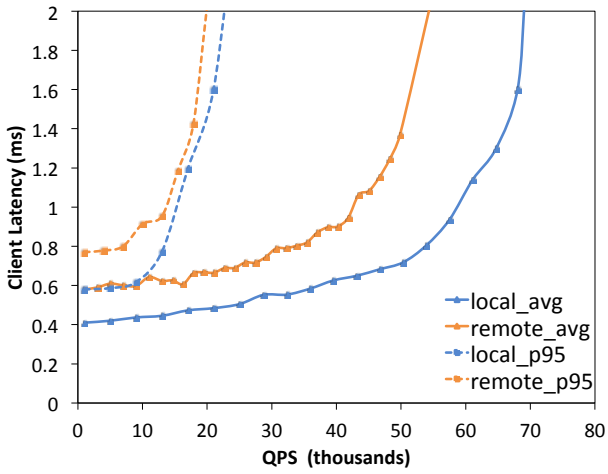
Workload	Read IOPS/TB	Read size (kB)	Write IOPS/TB	Write size (kB)
A	2k - 10k	10	100	500
B	2k - 6k	50	1000	700
C	2k - 4k*	15	500	2000
D	2k - 4k	25	150	500

*75K read IOPS are also observed 1% of the time

[3] Flow Director [32], an advanced feature that establishes a unique association between a flow and a CPU core with the consuming network application, would automatically steer a tenant's network traffic to the core to which we pinned the associated `istd` process.



(a) CDF of client latency in an unloaded system. Remote access adds 260μs to p95 latency, an acceptable overhead for our target applications.



(b) For a single SSDB server, remote vs. local Flash decreases client QPS by 20% on average. The percentage overhead is 10% for requests at the tail.

Figure 4: Client application performance with local vs. remote Flash (with optimizations enabled).

hosted on each datastore server uses 16 reader threads and 16 writer threads to manage a 24GB RocksDB database of 2.4 million key-value pairs. The RocksDB-managed block cache has a 10–15% hit rate. RocksDB workloads also typically benefit from the OS page cache. To avoid the unpredictability of the OS page cache, we use Linux containers (`cgroups`) to limit the memory available for page caching to only 4GB. Hence, we see a higher number of Flash accesses than a regular deployment, which is pessimistic in terms of the overhead induced by remote Flash accesses.

The main performance metric for our model workloads is the number of queries per second (QPS) processed. In particular, the number of `get` requests per second is an important performance metric, since read operations are blocking whereas writes are batched in RocksDB's memory buffer and return asynchronously to the client before being persisted, as is common in LSM database architectures. Although the rate or latency of write operations does not directly determine application performance, write operations are still important to model since they interfere with read operations at the Flash device and can significantly increase request queuing delays. From a storage perspective, we aim to maximize random read IOPS for these applications, so we apply the optimizations presented in Section 3.3 on the remote Flash server and show their impact on end-to-end performance.

4.2 End-to-end Application Performance

We quantify the performance overheads associated with disaggregating Flash on datastore servers, from the end-to-end perspective of the application tier. We first measure performance with a datastore tier consisting of a single server with local versus remote Flash. We then scale the datastore tier to multiple servers and compare end-to-end application performance in the scenario where each datastore server accesses its own local Flash device and the scenario where datastore servers access remote Flash on a *shared* server in the Flash tier. Unless otherwise noted, latency refers to read latency measured by the application tier and QPS refers to the queries per second (both read and write).

Single Datastore Server:

Latency: To evaluate the impact of remote Flash on end-to-end request latency, we measure round-trip read latency for requests issued to a single, unloaded datastore server. Figure 4a shows the cumulative distribution of end-to-end read latency with local and remote Flash on datastore servers. The overhead of remote Flash access on the 95th percentile latency observed by the application tier is 260μs. The inherent latency overhead of iSCSI is well within the acceptable range for our target applications, since latency SLAs are between 5 to 10ms rather than the sub-millisecond range. We discuss throughput-induced latency below.

Throughput: We sweep the number of queries per second issued by the application tier and plot the average and tail latency in Figure 4b. On average, end-to-end request latency with remote Flash saturates at 80% of the throughput achieved with local Flash on datastore servers. In both the local and remote Flash scenarios, latency saturates due to high CPU utilization on datastore servers. A 20% average drop in QPS with remote versus local Flash may sound like a high overhead, but with a disaggregated architecture, we can compensate for throughput loss by independently scaling datastore server CPUs from Flash. We analyze the performance-cost trade-off in detail in Section 4.3 to show that disaggregation can still lead to significant resource savings even with these performance overheads.

Online services are more likely to be provisioned based on tail, rather than average, performance requirements [42]. Figure 4b shows that tail latency (we plot the 95th percentile) is not as heavily impacted by remote Flash access as average latency. This is because read requests at the tail are requests serviced at the Flash during a RocksDB compaction or memtable flush which generates large write IOs, creating backlog in Flash queues. Read requests caught behind write operation at the Flash device experience high queuing delay because Flash writes take significantly longer than reads. Read-write performance asymmetry is a well-known characteristic of Flash storage technology [13, 57]. For read requests interfering with writes, iSCSI processing introduces a lower percentage overhead of 10%.

Impact of optimizations: Figure 5 shows the impact of the optimizations discussed in Section 3.3 on end-to-end application performance. The baseline in Figure 5 shows the average latency versus throughput curve with a single iSCSI process setup. Using 6 iSCSI processes instead of 1 allows the application to achieve over 3× higher QPS before average end-to-end latency spikes beyond 1.5 ms (see the curve labeled multi-process). Application QPS is 14% higher with the 6 `istiod` process iSCSI configuration than the default 8 process configuration (not shown in the figure). Next, we apply network optimizations and steer interrupts to achieve over 24% increase in average throughput (assuming a 1.5 ms latency SLA). More specifically, enabling segmentation offloads on the NIC increases QPS by 7% and jumbo frames increases QPS by an additional 13%. Finally, interrupt affinity tuning increases end-to-end QPS by 4%. The dotted blue line represents the performance limit, measured with local Flash (this is the same as the solid blue curve in Figure 4b).

Sensitivity analysis: To understand how disaggregating Flash impacts a variety of key-value store applications, we sweep workload parameters in our setup. We model varying degrees of compute intensity for the datastore service and the serialization/deserialization tasks it performs by adding busy loop iterations in SSDB, which execute before requests are issued to RocksDB. Figure 6a plots peak QPS as a function of the number of busy loop iterations inserted in the SSDB wrapper, showing that iSCSI overheads become negligible for client performance as the compute intensity on datastore servers increases. For all other experiments in this paper, we do not insert busy cycles in SSDB. We also evaluate performance sensitivity to the percentage of write requests (the application's `get:set` ratio). As we increase the percentage of write requests, the peak QPS in Figure 6b increases because write operations return asynchronously (writes are batched in the RocksDB memtable buffer and do not propagate immediately to Flash). We find that irrespective of their `get:set` ratio, application clients observe roughly the same QPS degradation when the datastore server they communicate with has remote versus local Flash.

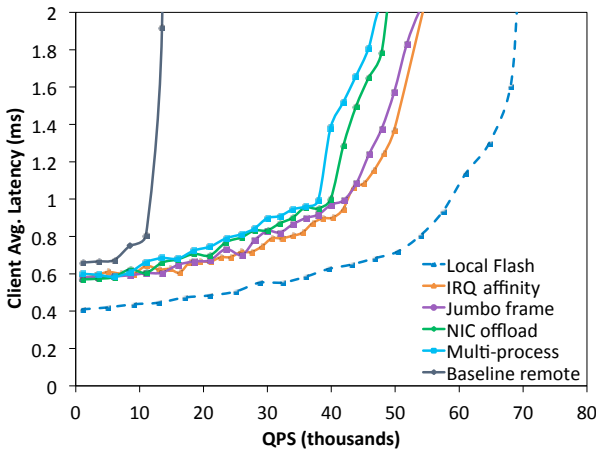
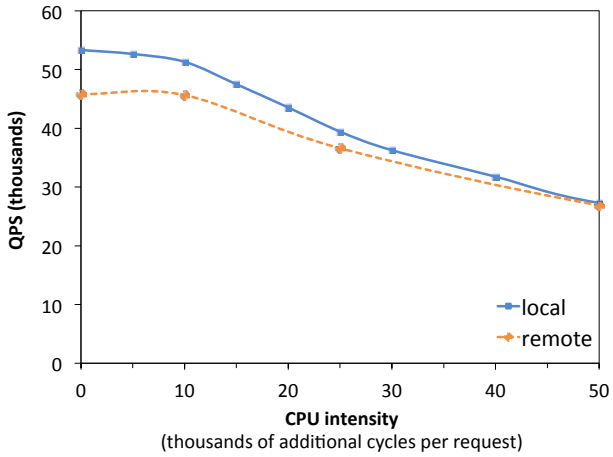
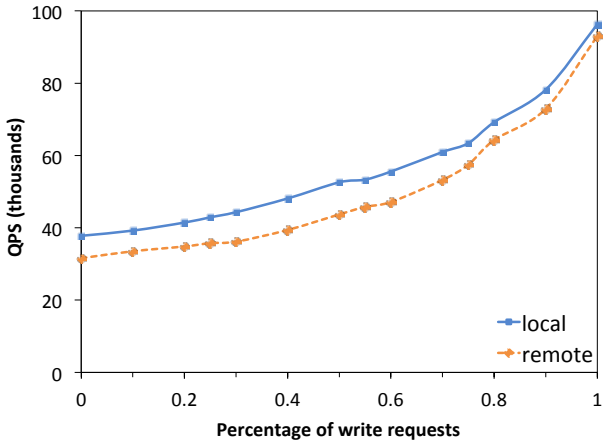


Figure 5: End-to-end impact of optimizations on average latency vs. throughput of key-value store application.

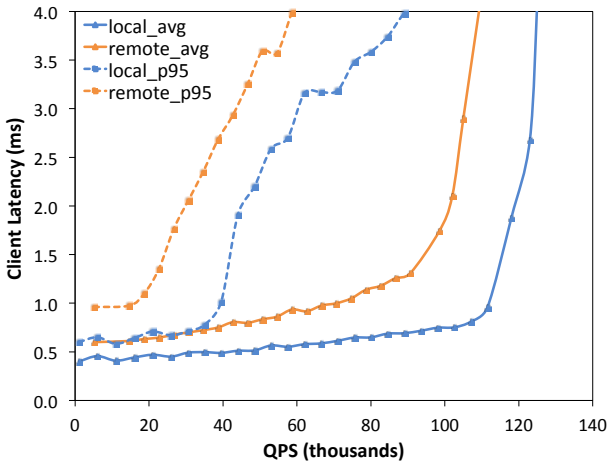


(a) As CPU intensity per request increases at the datastore server, the performance gap between local and remote Flash becomes negligible.

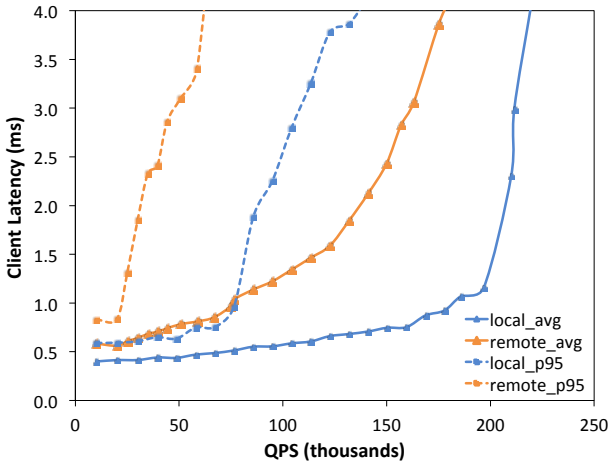


(b) The read/write ratio of a client application does not significantly affect the overhead it observes with remote vs. local Flash.

Figure 6: Workload parameter sensitivity sweep.



(a) 2 SSDB servers sharing remote Flash achieve 80% of the QPS of 2 SSDB servers with local Flash and utilize half the Flash resources. Increased write interference on shared Flash degrades tail read performance by 25%.



(b) With 3 SSDB servers sharing remote Flash, average QPS still degrades by 20% with remote Flash but tail performance degrades even further (55%) due to increased write interference.

Figure 7: Client application performance on shared remote Flash server (with optimizations enabled).

Multiple Datastore Servers: Next, we scale the number of servers in the datastore tier that share a remote Flash server and evaluate the impact of sharing Flash on application performance. Figure 7a shows the aggregate performance of 2 datastore servers, where *local* means each datastore server has its own direct-attached Flash drive and *remote* means each datastore server accesses separate partitions of a shared remote Flash server. Doubling the number of servers in the datastore tier doubles the application's aggregate average QPS. Even though the two datastore servers share a remote Flash server (thus using half the Flash resources compared to the local Flash deployment), the throughput overhead of remote Flash access remains 20% on average. Thus, we observe the same average throughput overhead for remote Flash with and without sharing.

However, the performance of tail latency requests degrades more noticeably when sharing remote Flash compared to the single-tenant scenario. Performance degradation at the tail is due to higher write throughput at the Flash. When a Flash server is shared between multiple datastore servers, the Flash device handles writes associated with write buffer flushes and database compactions of all tenants. When more write IOs are queued at the Flash device, read operations are more likely to get caught behind slow writes. This effect is even more pronounced in Figure 7b where a remote Flash server is shared between 3 datastore servers. To confirm that the tail effect is indeed due to write interference, we ran experiments with a SSDB server sharing remote Flash with 2 read-only tenants and found that tail latency did not degrade (performance was similar to the single-tenant experiment in Figure 4b).

In summary, we have shown that the latency overhead of iSCSI processing is not a concern for end-to-end performance of our target application. Processing overhead has a more substantial impact on throughput. On average, for our target application, a datastore server accessing remote Flash supports 80% of the QPS it could with local Flash. We will show in the next section how we can make up for this throughput loss by scaling the number of datastore servers independently from Flash with a disaggregated architecture. Our observation that sharing Flash among tenants with bursty write workloads degrades tail read latency has important implications for the control plane. The coordination manager responsible for allocating Flash capacity and IOPS in the datacenter needs to carefully select compatible tenants to share Flash based on workload write patterns and tail read latency SLAs, as we discuss in Section 5.2.

Resource overhead: In addition to performance overhead, we also evaluate the resource overhead for disaggregated Flash, namely the number of cores required for iSCSI protocol processing at a remote Flash server. In Section 3.3, for our IO-intensive microbenchmark, we found it best to allocate at least 1 CPU core per tenant. However, for our workloads modeled based on real datacenter applications, we find that a Flash tier server with a single CPU core[4] can support up to 3 datastore tenants running at peak load. With 4 datastore tenants per core on the Flash server, we observe that CPU utilization on the Flash server becomes a bottleneck and application performance drops. Application performance recovers with 4 datastore tenants if we allocate an additional core on the Flash server.

[4] We turn cores on and off on our Linux server by setting the value in `/sys/devices/system/cpu/cpuX/online` to 1 and 0, respectively.

4.3 Disaggregation Benefits

Our evaluation thus far has focused on the overheads of disaggregation. In this section, we model, to a first-order approximation, the impact of disaggregation on resource efficiency to show when the *benefits* of independent resource scaling outweigh the overheads of remote access to Flash.

For this part of our analysis, we assume that disaggregated Flash is on *dedicated storage servers*, which only process network storage requests and do not host local Flash-based applications. For simplicity, we also assume that resource costs scale linearly. In reality, CPU cost may increase exponentially with performance for high-end servers and Flash is often cheaper per GB when purchased with larger capacity. Our cost model is intended for a first-order analysis.

Cost model: We formulate the resource cost of hosting an application with a target performance QPS_t, a data set of size GB_t, and an IOPS requirement IOPS_t. C_{direct} and C_{disagg} represent the capital cost for a cluster of servers with direct-attached Flash and disaggregated Flash, respectively:

$$C_{direct} = \max\left(\frac{\text{GB}_t}{\text{GB}_s}, \frac{\text{IOPS}_t}{\text{IOPS}_s}, \frac{\text{QPS}_t}{\text{QPS}_s}\right) \cdot \left(f + c\right) \quad (1)$$

$$C_{disagg} = \max\left(\frac{\text{GB}_t}{\text{GB}_s}, \frac{\text{IOPS}_t}{\text{IOPS}_s}\right) \cdot \left(f + \delta\right) + \left(\frac{\text{QPS}_t}{\text{QPS}_s}\right) c \quad (2)$$

where:

f: cost of Flash on a server
c: cost of CPU, RAM and NIC on datastore server
δ: cost of CPU, RAM and NIC on Flash tier server, i.e. resource "tax" for disaggregation
x_s: x provided by a single server, $x = \{\text{GB, IOPS, QPS}\}$
x_t: x required in total for the application

With a direct-attached Flash server architecture, servers consist of CPU, memory and Flash resources. The number of servers deployed depends on the maximum of the compute, capacity and IOPS requirements of the application. In contrast, with a disaggregated Flash server architecture, the amount of CPU and memory deployed depends solely on the compute and memory requirements of the application while the amount of Flash deployed depends solely on the Flash capacity and IOPS requirements of the application. The costs C_{direct} and C_{disagg} differ significantly when there is a substantial difference between the application's compute and storage requirements, $\frac{\text{QPS}_t}{\text{QPS}_s}$ and $\max\left(\frac{\text{GB}_t}{\text{GB}_s}, \frac{\text{IOPS}_t}{\text{IOPS}_s}\right)$ respectively.

Example calculation for target application: The applications we model in this study are compute-intensive rather than capacity or IOPS-limited. Thus the number of datastore servers deployed to host an application with target performance QPS_t depends on the number of queries handled by CPUs per server, QPS_s. We showed in Section 4.2 that datastore servers accessing remote Flash have a 20% lower QPS_s due to iSCSI overhead, compared to datastore servers with direct-attached Flash. To make our example calculation of server costs with Equations 1 and 2 more concrete, we assume our application has target performance QPS_t = 10M and a data set of size 100TB. We assume 1.2TB capacity PCIe Flash drives. For the purposes of this study, we consulted online retail catalogs, which provided the following ballpark costs for CPU, memory, PCIe Flash resources on servers: $c = \$2,400$, $f = \$2,400$, and $\delta = \$50$/tenant since we assume 3 tenants share each \$100 CPU core on a remote Flash server and we add some cost overhead for the NIC and memory. Note that memory requirements on the remote Flash server are low since the block-IO protocol does not rely on caching. Our example resource costs align with Cully et al.'s observations that PCIe Flash often costs as much as (or even more than) the rest of the components on a storage server [15]. Substituting these costs and our QPS performance measurements from Section 4.2 into Equations 1 and 2, we have:

$$C_{direct} = \frac{10 \times 10^6}{50 \times 10^3}(2400 + 2400) = 960,000$$

$$C_{disagg} = \frac{100 \times 10^9}{1.2 \times 10^9}(2400 + 3(50)) + \frac{10 \times 10^6}{40 \times 10^3}2400 = 812,500$$

Cost benefits of disaggregation: The example calculation above shows a 14% cost benefit using disaggregated Flash versus direct-attached Flash for an application at a snapshot in time. We expect the true benefits of disaggregation to become apparent as application requirements scale over time. When CPU intensity scales differently than storage requirements for an application, the flexibility to scale CPU and memory on datastore servers independently from Flash storage can lead to long-term cost benefits.

Figure 8 plots the savings of disaggregating Flash as a function of the scaling factor for an application's storage capacity requirements on the horizontal axis and compute requirements on the vertical axis. The origin represents a baseline server deployment for an application, where Flash and CPU utilization are balanced. Figure 8 highlights the scenarios where disaggregation is most beneficial: 1) when compute intensity scales at a higher rate than Flash capacity (top left corner) because we can deploy compute servers and share Flash among them, or 2) when capacity scales at a higher rate than compute intensity (bottom right corner) because then we can deploy more Flash without needing to add expensive CPU and memory resources. Disaggregating Flash does not make sense when compute and storage requirements remain approximately balanced (diagonal line from origin), since then we pay for the overheads of disaggregation without reaping the benefits of better resource utilization compared to a direct-attached Flash architecture.

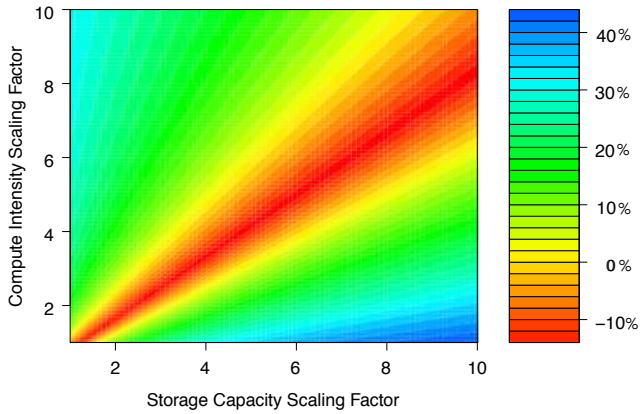


Figure 8: Percentage resource savings of disaggregated Flash as a function of compute intensity and storage capacity requirement scaling. The origin represents an architecture with balanced utilization of CPU and direct-attached Flash. Disaggregation can be cost-effective when compute and storage scale at different rates.

The percentage resource savings in Figure 8 assume that the cost of PCIe Flash and other resources on a datastore server are approximately equal, as in the example calculation above where $f \approx c$. We perform a sensitivity analysis to understand how resource savings vary with different assumptions about the cost ratio between Flash and other server resources. If we assume the price of Flash decreases in comparison to CPU and memory, disaggregation becomes even more beneficial for workloads with high storage requirements since the flexibility to deploy Flash without adding unnecessary CPU and memory is even more advantageous when these resources cost more in comparison to Flash. On the other hand, if we assume PCIe Flash is more expensive than compute and memory resources on a datastore server, then compute-intensive applications with low storage requirements benefit more from disaggregation. For example, doubling the price of PCIe Flash in our calculations leads to 50% resource savings in the top left corner of Figure 8. Finally, if we assume a higher δ, the resource tax associated with disaggregation, then hosting applications with balanced CPU and storage requirements becomes around 20% more expensive with disaggregated Flash architectures compared to local Flash architectures (as opposed to 12% in Figure 8) but applications with imbalanced resource requirements still have similar percentage resource savings.

5. Discussion

We summarize key insights from our study to guide future work on remote Flash, discussing implications for both the dataplane and control plane.

5.1 Dataplane implications

Our analysis of remote Flash in the context of a datacenter application using a traditional iSCSI storage stack reveals opportunities to improve the storage dataplane.

Reduce overhead in application software: Our analysis showed that most of the overhead associated with accessing Flash remotely using a heavy-weight protocol like iSCSI is masked by CPU overhead in upper layers of datacenter application software. For our target application, remote versus local access to Flash decreases end-to-end throughput by 20%. However, for a 4kB IO intensive microbenchmark, a single tenant experiences 70% overhead for remote access. The datacenter application experiences 50% less overhead because it saturates CPU resources even when accessing local Flash. This shows it is important to reduce compute intensity in datastore processing and serialization for inter-tier communication with generic RPC frameworks.

Reduce overhead in network storage stack: There is also significant overhead to cut down at the remote storage protocol layer. For our target application, the throughput overhead with iSCSI is 20%. However, for applications that issue more than 10,000s of IOPS, the CPU intensity of iSCSI introduces more noticeable performance overhead. For example, we observed 70% throughput overhead for our IO intensive microbenchmark which can saturate local PCIe Flash with 100,000s of IOPS. In addition to throughput overhead, there is latency overhead. The 260μs that iSCSI adds to tail latency is acceptable for our application which has latency SLAs $>$ 1ms, but the overhead is too high for applications sensitive to latency at the *microsecond* scale.

One approach for reducing CPU overheads for remote Flash access is to optimize network processing either within the kernel or in a user-level stack. For example, mTCP is a user-level TCP/IP stack that improves network throughput compared to Linux by batching events and using lock-free, per-core data structures [36]. Other user-level network stacks implement similar techniques [48, 67]. Network processing can be optimized within the kernel too. For example, IX provides a protected data plane that achieves higher throughput and lower latency than mTCP [10]. Key design principles in IX that enable high performance are run to completion, adaptive batching, zero-copy APIs and flow-based, synchronization-free processing. Other kernel-level efforts apply similar techniques [27, 58]. It is worth exploring the benefits of these techniques in the context of remote storage.

Another approach for reducing the CPU overhead of remote Flash access is to use a lightweight network protocol. RDMA-based protocols such as RoCE reduce CPU overhead by offloading protocol processing to hardware and avoiding data copies to and from memory buffers [49]. However, RDMA requires more expensive network hardware (RDMA-capable NICs) and protocols commonly assume a lossless fabric, thus limiting the scale of disaggregation.

5.2 Control plane implications

Our study has focused on the dataplane for remote access to Flash. We discuss implications for resource management by the control plane at both the cluster and storage node level, which we leave to future work.

Cluster resource management: Given an application's requirements for storage capacity, IOPS, and latency, the control plane must decide which storage servers to allocate. This is a typical bin packing problem that all cluster managers address [16, 29, 62, 71]. Selecting specific workloads to collocate on specific servers is a multi-diemnsional problem with constantly evolving constraints as application requirements vary over time (as shown in Figure 1). Assigning resources to stateful applications is particularly challenging since correcting mistakes is expensive. Our study showed that to avoid high tail read latency, the control plane should consider each workload's read and write patterns when assigning applications to shared storage servers.

Storage node resource isolation: While the cluster manager should try to assign applications with compatible IO patterns to share Flash, mechanisms for performance isolation and quality of servers at the storage node are also important for improving performance in a disaggregated Flash deployment. Previous work in shared storage management has shown the benefits of techniques such as time-slicing, rate limiting, request amortization, and QoS-aware scheduling with feedback [4, 24, 54, 64, 65, 73, 74]. It is worth revisiting QoS mechanisms in the context of modern, multi-queue PCIe Flash devices. For example, managing multiple hardware Flash queues exposed by the NVMe interface may be useful for enforcing priorities and performance isolation.

6. Conclusion

PCIe-based Flash is increasingly being deployed in datacenter servers to provide high IOPS. However, the capacity and bandwidth of these high-performance devices are commonly underutilized due to imbalanced resource requirements across applications and over time. In this work, we have examined disaggregating Flash as a way of improving resource efficiency and dealing with Flash overprovisioning. We demonstrated how to tune remote access to Flash over commodity networks through techniques like interrupt steering. We also analyzed the end-to-end performance implications of remote Flash for workloads sampled from real datacenter applications. Our analysis showed that although remote Flash access introduces a 20% throughput drop at the application level, disaggregation allows us to make up for these overheads by independently scaling CPU and Flash resources in a cost effective manner.

Acknowledgments

We thank Christina Delimitrou, Heiner Litz, and the anonymous reviewers for their valuable feedback. We also thank David Capel and Igor Canadi for answering our questions about the Facebook workloads modeled in this study. This work is supported by Facebook, the Stanford Platform Lab, and NSF grant CNS-1422088. Ana Klimovic is supported by a Microsoft Research PhD Fellowship.

References

[1] Amazon. Amazon Elastic Block Store . `https://aws.amazon.com/ebs/`, 2016.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. of USENIX Hot Topics in Operating Systems*, HotOS'13, pages 12–12, 2011.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proc. of ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 1–14. ACM, 2009.

[4] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 233–248, Oct. 2014.

[5] Apache Software Foundation. Apache Thrift. `https://thrift.apache.org`, 2014.

[6] Avago Technologies. Storage and PCI Express – A Natural Combination. `http://www.avagotech.com/applications/datacenters/enterprise-storage`, 2015.

[7] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'12, pages 1–1, 2012.

[8] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 351–365, Oct. 2014.

[9] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[10] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Oct. 2014.

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

[12] Chelsio Communications. NVM Express over Fabrics. `http://www.chelsio.com/wp-content/uploads/resources/NVM_Express_Over_Fabrics.pdf`, 2014.

[13] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192. ACM, 2009.

[14] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: a network stack for rack-scale computers. In *Proc. of ACM Con-*

ference on Special Interest Group on Data Communication, SIGCOMM '15, pages 551–564. ACM, 2015.

[15] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proc. of USENIX File and Storage Technologies (FAST 14)*, pages 17–31. USENIX, 2014.

[16] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIX, pages 127–144. ACM, 2014.

[17] Dell Inc. PowerEdge PCIe Express Flash SSD. `http://www.dell.com/learn/us/en/04/campaigns/poweredge-express-flash`, 2015.

[18] Facebook Inc. Open Compute Project. `http://www.opencompute.org/projects`, 2015.

[19] Facebook Inc. RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org`, 2015.

[20] Fusion IO. Atomic Series Server Flash. `http://www.fusionio.com/products/atomic-series`, 2015.

[21] S. Ghemawat and J. Dean. LevelDB. `https://github.com/google/leveldb`, 2014.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[23] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers`, 2015.

[24] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proc. of USENIX File and Storage Technologies*, FAST '09, pages 85–98, 2009.

[25] J. Hamilton. Keynote: Internet-scale service infrastructure efficiency. In *Proc. of International Symposium on Computer Architecture, ISCA '09*, June 2009.

[26] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proc. of ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7. ACM, 2013.

[27] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 135–148, 2012.

[28] HGST. LinkedIn scales to 200 million users with PCIe Flash storage from HGST. `https://www.hgst.com/sites/default/files/resources/LinkedIn-Scales-to-200M-Users-CS.pdf`, 2014.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'11, pages 295–308, 2011.

[30] HP. Moonshot system. `http://www8.hp.com/us/en/products/servers/moonshot/`, 2015.

[31] ideawu. SSDB with RocksDB. `https://github.com/ideawu/ssdb-rocks`, 2014.

[32] Intel. Intel Ethernet Flow Director. `http://www.intel.com/content/www/us/en/ethernet-products/ethernet-flow-director-video.html`, 2016.

[33] Intel Corp. Intel Rack Scale Architecture Platform. `http://www.intel.com/content/dam/www/public/us/en/documents/guides/rack-scale-hardware-guide.pdf`, 2015.

[34] Intel Corp. Intel Solid-State Drive DC P3600 Series. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3600-spec.pdf`, 2015.

[35] Jens Axboe. Flexible IO tester (FIO). `http://git.kernel.dk/?p=fio.git;a=summary`, 2015.

[36] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level tcp stack for multicore systems. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[37] A. Joglekar, M. E. Kounavis, and F. L. Berry. A scalable and high performance software iSCSI implementation. In *In Proc. of USENIX File and Storage Technologies.*, pages 267–280, 2005.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, Aug. 2014.

[39] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. In *Proc. of Annual International Symposium on Computer Architecture, ISCA '15*, pages 158–169, 2015.

[40] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92. ACM, 1996.

[41] J. Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[42] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. of European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14. ACM, 2014.

[43] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009)*, pages 267–278, 2009.

[44] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012*, pages 189–200, 2012.

[45] LinkedIn Inc. Project Voldemort: A distributed key-value storage system. `http://www.project-voldemort.com/voldemort`, 2015.

[46] C. Loboz. Cloud resource usage-heavy tailed distributions invalidating traditional capacity planning models. *Journal of Grid Computing*, 10(1):85–108, 2012.

[47] Y. Lu and D. Du. Performance study of iSCSI-based storage subsystems. *Communications Magazine, IEEE*, 41(8):76–82, Aug 2003.

[48] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proc. of ACM SIGCOMM*, SIGCOMM'14, pages 175–186, 2014.

[49] Mellanox Technologies. RoCE in the Data Center. `http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf`, 2014.

[50] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer Publishing Company, Incorporated, 2012.

[51] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'14, pages 257–273, Apr. 2014.

[52] Microsoft. Introduction to Receive Side Scaling. `https://msdn.microsoft.com/library/windows/hardware/ff556942.aspx`, 2016.

[53] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proc. of European Conference on Computer Systems*, EuroSys '09, pages 145–158. ACM, 2009.

[54] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proc. of European Conference on Computer Systems*, EuroSys '10, pages 237–250. ACM, 2010.

[55] NVM Express Inc. NVM Express: the optimized PCI Express SSD interface. `http://www.nvmexpress.org`, 2015.

[56] J. Ouyang, S. Lin, J. Song, Z. Hou, Y. Wang, and Y. Wang. SDF: software-defined flash for web-scale internet storage systems. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIX, pages 471–484, 2014.

[57] S. Park and K. Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proc. of USENIX File and Storage Technologies*, FAST'12, page 13, 2012.

[58] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proc. of ACM European Conference on Computer Systems*, EuroSys'12, pages 337–350. ACM, 2012.

[59] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy. A performance comparison of NFS and iSCSI for IP-networked storage. In *In Proc. of USENIX File and Storage Technologies.*, pages 101–114, 2004.

[60] R. Sandberg. Design and implementation of the Sun network filesystem. In *In Proc. of USENIX Summer Conference.*, pages 119–130. 1985.

[61] Satran, et al. Internet Small Computer Systems Interface (iSCSI). `https://www.ietf.org/rfc/rfc3720.txt`, 2004.

[62] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems*, EuroSys'13, pages 351–364, 2013.

[63] SeaMicro. SM15000 fabric compute systems. `http://www.seamicro.com/sites/default/files/SM15000_Datasheet.pdf`, 2015.

[64] D. Shue and M. J. Freedman. From application requests to virtual iops: provisioned key-value storage with libra. In *Proc. of European Conference on Computer Systems*, EuroSys'14, pages 17:1–17:14, 2014.

[65] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 349–362, 2012.

[66] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. of IEEE Mass Storage Systems and Technologies*, MSST '10, pages 1–10. IEEE Computer Society, 2010.

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

[68] M. Stokely, A. Mehrabian, C. Albrecht, F. Labelle, and A. Merchant. Projecting disk usage based on historical trends in a cloud environment. In *ScienceCloud Proc. of International Workshop on Scientific Cloud Computing*, pages 63–70, 2012.

[69] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh. Secure I/O device sharing among virtual machines on multiple hosts. In *Proc. of International Symposium on Computer Architecture*, ISCA '13, pages 108–119. ACM, 2013.

[70] M. Uysal, A. Merchant, and G. A. Alvarez. Using MEMS-based storage in disk arrays. In *Proc. of USENIX File and Storage Technologies*, FAST'03, pages 7–7, 2003.

[71] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. of European Conference on Computer Systems*, EuroSys'15, 2015.

[72] VMware. Virtual SAN. `https://www.vmware.com/products/virtual-san`, 2016.

[73] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proc. of USENIX File and Storage Technologies*, FAST '07, pages 5–5, 2007.

[74] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proc. of ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14. ACM, 2012.

[75] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proc. of USENIX Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 4–4, 2005.

[76] D. Xinidis, A. Bilas, and M. D. Flouris. Performance evaluation of commodity iSCSI-based storage systems. In *Proc. of IEEE/NASA Goddard Mass Storage Systems and Technologies*, MSST '05, pages 261–269. IEEE Computer Society, 2005.