Strata: Scalable High-Performance Storage on Virtualized Non-volatile Memory

Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffrey Lefebvre, Daniel Ferstay, and Andrew Warfield

Coho Data

{firstname.lastname}@cohodata.com

Abstract

Strata is a commercial storage system designed around the high performance density of PCIe flash storage. We observe a parallel between the challenges introduced by this emerging flash hardware and the problems that were faced with underutilized server hardware about a decade ago. Borrowing ideas from hardware virtualization, we present a novel storage system design that partitions functionality into an address virtualization layer for high performance network-attached flash, and a hosted environment for implementing scalable protocol implementations. Our system targets the storage of virtual machine images for enterprise environments, and we demonstrate dynamic scale to over a million IO operations per second using NFSv3 in 13u of rack space, including switching.

1 Introduction

Flash-based storage devices are fast, expensive and demanding: a single device is capable of saturating a 10Gb/s network link (even for random IO), consuming significant CPU resources in the process. That same device may cost as much as (or more than) the server in which it is installed[1]. The cost and performance characteristics of fast, non-volatile media have changed the calculus of storage system design and present new challenges for building efficient and high-performance datacenter storage.

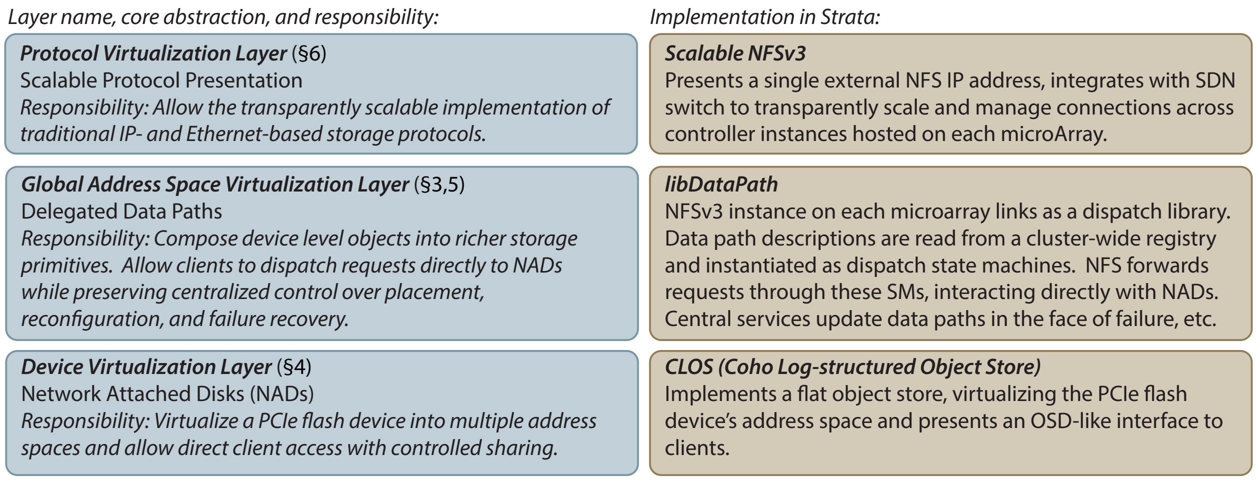
This paper describes the architecture of a commercial flash-based network-attached storage system, built using commodity hardware. In designing the system around PCIe flash, we begin with two observations about the effects of high-performance drives on large-scale storage systems. First, these devices are fast enough that in most environments, many concurrent workloads are needed to fully saturate them, and even a small degree of processing overhead will prevent full utilization. Thus, we must change our approach to the media from *aggregation* to *virtualization*. Second, aggregation is still necessary to achieve properties such as redundancy and scale. However, it must avoid the performance bottleneck that would result from the monolithic controller approach of a traditional storage array, which is designed around the obsolete assumption that media is the slowest component in the system. Further, to be practical in existing datacenter environments, we must remain compatible with existing client-side storage interfaces and support standard enterprise features like snapshots and deduplication.

In this paper we explore the implications of these two observations on the design of a scalable, high-performance NFSv3 implementation for the storage of virtual machine images. Our system is based on the building blocks of PCIe flash in commodity x86 servers connected by 10 gigabit switched Ethernet. We describe two broad technical contributions that form the basis of our design:

1. A delegated mapping and request dispatch interface from client data to physical resources through *global data address virtualization*, which allows clients to directly address data while still providing the coordination required for online data movement (e.g., in response to failures or for load balancing).
2. SDN-assisted *storage protocol virtualization* that allows clients to address a single virtual protocol gateway (e.g., NFS server) that is transparently scaled out across multiple real servers. We have built a scalable NFS server using this technique, but it applies to other protocols (such as iSCSI, SMB, and FCoE) as well.

At its core, Strata uses device-level object storage and dynamic, global address-space virtualization to achieve a clean and efficient separation between control and data paths in the storage system. Flash devices are split into

[1]Enterprise-class PCIe flash drives in the 1TB capacity range currently carry list prices in the range of $3-5K USD. Large-capacity, high-performance cards are available for list prices of up to $160K.



Layer name, core abstraction, and responsibility:	*Implementation in Strata:*
Protocol Virtualization Layer (§6) Scalable Protocol Presentation *Responsibility: Allow the transparently scalable implementation of traditional IP- and Ethernet-based storage protocols.*	***Scalable NFSv3*** Presents a single external NFS IP address, integrates with SDN switch to transparently scale and manage connections across controller instances hosted on each microArray.
Global Address Space Virtualization Layer (§3,5) Delegated Data Paths *Responsibility: Compose device level objects into richer storage primitives. Allow clients to dispatch requests directly to NADs while preserving centralized control over placement, reconfiguration, and failure recovery.*	***libDataPath*** NFSv3 instance on each microarray links as a dispatch library. Data path descriptions are read from a cluster-wide registry and instantiated as dispatch state machines. NFS forwards requests through these SMs, interacting directly with NADs. Central services update data paths in the face of failure, etc.
Device Virtualization Layer (§4) Network Attached Disks (NADs) *Responsibility: Virtualize a PCIe flash device into multiple address spaces and allow direct client access with controlled sharing.*	***CLOS (Coho Log-structured Object Store)*** Implements a flat object store, virtualizing the PCIe flash device's address space and presents an OSD-like interface to clients.

Figure 1: Strata network storage architecture.

virtual address spaces using an object storage-style interface, and clients are then allowed to directly communicate with these address spaces in a safe, low-overhead manner. In order to compose richer storage abstractions, a global address space virtualization layer allows clients to aggregate multiple per-device address spaces with mappings that achieve properties such as striping and replication. These delegated address space mappings are coordinated in a way that preserves direct client communications with storage devices, while still allowing dynamic and centralized control over data placement, migration, scale, and failure response.

Serving this storage over traditional protocols like NFS imposes a second scalability problem: clients of these protocols typically expect a single server IP address, which must be dynamically balanced over multiple servers to avoid being a performance bottleneck. In order to both scale request processing and to take advantage of full switch bandwidth between clients and storage resources, we developed a *scalable protocol presentation layer* that acts as a client to the lower layers of our architecture, and that interacts with a software-defined network switch to scale the implementation of the protocol component of a storage controller across arbitrarily many physical servers. By building protocol gateways as clients of the address virtualization layer, we preserve the ability to delegate scale-out access to device storage without requiring interface changes on the end hosts that consume the storage.

2 Architecture

The performance characteristics of emerging storage hardware demand that we completely reconsider storage architecture in order to build scalable, low-latency shared persistent memory. The reality of deployed applications is that interfaces must stay exactly the same in order for a storage system to have relevance. Strata's architecture aims to take a step toward the first of these goals, while keeping a pragmatic focus on the second.

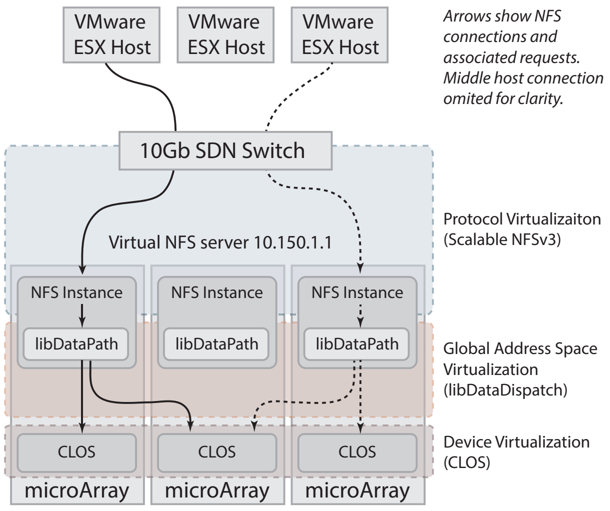
Figure 1 characterizes the three layers of Strata's architecture. The goals and abstractions of each layer of the system are on the left-hand column, and the concrete embodiment of these goals in our implementation is on the right. At the base, we make devices accessible over an object storage interface, which is responsible for virtualizing the device's address space and allowing clients to interact with individual virtual devices. This approach reflects our view that system design for these storage devices today is similar to that of CPU virtualization ten years ago: devices provide greater performance than is required by most individual workloads and so require a lightweight interface for controlled sharing in order to allow multi-tenancy. We implement a per-device object store that allows a device to be virtualized into an address space of 2^{128} sparse objects, each of which may be up to 2^{64} bytes in size. Our implementation is similar in intention to the OSD specification, itself motivated by network attached secure disks [17]. While not broadly deployed to date, device-level object storage is receiving renewed attention today through pNFS's use of OSD as a backend, the NVMe *namespace* abstraction, and in emerging hardware such as Seagate's Kinetic drives [37]. Our object storage interface as a whole is not a significant technical contribution, but it does have some notable interface customizations described in Section 4. We refer to this layer as a *Network Attached Disk*, or *NAD*.

The middle layer of our architecture provides a global address space that supports the efficient composition of

IO processors that translate client requests on a *virtual object* into operations on a set of NAD-level *physical objects*. We refer to the graph of IO processors for a particular virtual object as its *data path*, and we maintain the description of the data path for every object in a global *virtual address map*. Clients use a dispatch library to instantiate the processing graph described by each data path and perform direct IO on the physical objects at the leaves of the graph. The virtual address map is accessed through a coherence protocol that allows central services to update the data paths for virtual objects while they are in active use by clients. More concretely, data paths allow physical objects to be composed into richer storage primitives, providing properties such as striping and replication. The goal of this layer is to strike a balance between scalability and efficiency: it supports direct client access to device-level objects, without sacrificing central management of data placement, failure recovery, and more advanced storage features such as deduplication and snapshots.

Finally, the top layer performs *protocol virtualization* to allow clients to access storage over standard protocols (such as NFS) without losing the scalability of direct requests from clients to NADs. The presentation layer is tightly integrated with a 10Gb software-defined Ethernet switching fabric, allowing external clients the illusion of connecting to a single TCP endpoint, while transparently and dynamically balancing traffic to that single IP address across protocol instances on all of the NADs. Each protocol instance is a thin client of the layer below, which may communicate with other protocol instances to perform any additional synchronization required by the protocol (e.g., to maintain NFS namespace consistency).

The mapping of these layers onto the hardware that our system uses is shown in Figure 2. Requests travel from clients into Strata through an OpenFlow-enabled switch, which dispatches them according to load to the appropriate protocol handler running on a *MicroArray (μArray)* — a small host configured with flash devices and enough network and CPU to saturate them, containing the software stack representing a single NAD. For performance, each of the layers is implemented as a library, allowing a single process to handle the flow of requests from client to media. The NFSv3 implementation acts as a client of the underlying dispatch layer, which transforms requests on virtual objects into one or more requests on physical objects, issued through function calls to local physical objects and by RPC to remote objects. While the focus of the rest of this paper is on this concrete implementation of scale-out NFS, it is worth noting that the design is intended to allow applications the opportunity to link directly against the same data path library that the NFS implementation uses, resulting in a multi-tenant, multi-



VMware ESX Host
VMware ESX Host
VMware ESX Host
Arrows show NFS connections and associated requests. Middle host connection omited for clarity.
10Gb SDN Switch
Virtual NFS server 10.150.1.1
Protocol Virtualizaiton (Scalable NFSv3)
NFS Instance
NFS Instance
NFS Instance
libDataPath
libDataPath
libDataPath
Global Address Space Virtualization (libDataDispatch)
CLOS
CLOS
CLOS
Device Virtualization (CLOS)
microArray
microArray
microArray

Figure 2: Hardware view of a Strata deployment

presentation storage system with a minimum of network and device-level overhead.

2.1 Scope of this Work

There are three aspects of our design that are not considered in detail within this presentation. First, we only discuss NFS as a concrete implementation of protocol virtualization. Strata has been designed to host and support multiple protocols and tenants, but our initial product release is specifically NFSv3 for VMware clients, so we focus on this type of deployment in describing the implementation. Second, Strata was initially designed to be a software layer that is co-located on the same physical servers that host virtual machines. We have moved to a separate physical hosting model where we directly build on dedicated hardware, but there is nothing that prevents the system from being deployed in a more co-located (or "converged") manner. Finally, our full implementation incorporates a tier of spinning disks on each of the storage nodes to allow cold data to be stored more economically behind the flash layer. However, in this paper we configure and describe a single-tier, all-flash system to simplify the exposition.

In the next sections we discuss three relevant aspects of Strata—address space virtualization, dynamic reconfiguration, and scalable protocol support—in more detail. We then describe some specifics of how these three components interact in our NFSv3 implementation for VM image storage before providing a performance evaluation of the system as a whole.

3 Data Paths

Strata provides a common library interface to data that underlies the higher-level, client-specific protocols described in Section 6. This library presents a notion of virtual objects, which are available cluster-wide and may comprise multiple physical objects bundled together for parallel data access, fault tolerance, or other reasons (e.g., data deduplication). The library provides a superset of the object storage interface provided by the NADs (Section 4), with additional interfaces to manage the placement of objects (and ranges within objects) across NADs, to maintain data invariants (e.g., replication levels and consistent updates) when object ranges are replicated or striped, and to coordinate both concurrent access to data and concurrent manipulation of the *virtual address maps* describing their layout.

To avoid IO bottlenecks, users of the data path interface (which may be native clients or protocol gateways such as our NFS server) access data directly. To do so, they map requests from virtual objects to physical objects using the virtual address map. This is not simply a pointer from a virtual object *(id, range)* pair to a set of physical object *(id, range)* pairs. Rather, each virtual range is associated with a particular *processor* for that range, along with processor-specific context. Strata uses a dispatch-oriented programming model in which a pipeline of operations is performed on requests as they are passed from an originating client, through a set of transformations, and eventually to the appropriate storage device(s). Our model borrows ideas from packet processing systems such as X-Kernel [19], Scout [25], and Click [21], but adapts them to a storage context, in which modules along the pipeline perform translations through a set of layered address spaces, and may fork and/or collect requests and responses as they are passed.

The dispatch library provides a collection of request processors, which can stand alone or be combined with other processors. Each processor takes a storage request (e.g., a read or write request) as input and produces one or more requests to its children. NADs expose isolated sparse objects; processors perform translations that allow multiple objects to be combined for some functional purpose, and present them as a single object, which may in turn be used by other processors. The idea of request-based address translation to build storage features has been used in other systems [24,35,36], often as the basis for volume management; Strata disentangles it from the underlying storage system and treats it as a first-class dispatch abstraction.

The composition of dispatch modules bears similarity to Click [21], but the application in a storage domain carries a number of differences. First, requests are generally acknowledged at the point that they reach a storage device, and so as a result they differ from packet forwarding logic in that they travel both down and then back up through a dispatch stack; processors contain logic to handle both requests *and* responses. Second, it is common for requests to be split or merged as they traverse a processor — for example, a replication processor may duplicate a request and issue it to multiple nodes, and then collect all responses before passing a single response back up to its parent. Finally, while processors describe fast, library-based request dispatching logic, they typically depend on additional facilities from the system. Strata allows processor implementations access to APIs for shared, cluster-wide state which may be used on a control path to, for instance, store replica configuration. It additionally provides facilities for background functionality such as NAD failure detection and response. The intention of the processor organization is to allow dispatch decisions to be pushed out to client implementations and be made with minimal performance impact, while still benefiting from common system-wide infrastructure for maintaining the system and responding to failures. The responsibilities of the dispatch library are described in more detail in the following subsections.

3.1 The Virtual Address Map

```
/objects/112:
    type=regular dispatch={object=111
                           type=dispatch}

/objects/111:
    type=dispatch
    stripe={stripecount=8 chunksize=524288
        0={object=103 type=dispatch}
        1={object=104 type=dispatch}}

/objects/103:
    type=dispatch
    rpl={policy=mirror storecount=2
            {storeid=a98f2... state=in-sync}
            {storeid=fc89f... state=in-sync}}
```

Figure 3: Virtual object to physical object range mapping

Figure 3 shows the relevant information stored in the virtual address map for a typical object. Each object has an identifier, a type, some type-specific context, and may contain other metadata such as cached size or modification time information (which is not canonical, for reasons discussed below).

The entry point into the virtual address map is a *regular* object. This contains no location information on its own, but delegates to a top-level *dispatch* object. In Figure 3, object 112 is a regular object that delegates to a dispatch processor whose context is identified by object 111 (the IDs are in reverse order here because the dispatch graph is created from the bottom up, but traversed from the top down). Thus when a client opens file 112, it instantiates a dispatcher using the data in object 111 as context. This context informs the dispatcher that it will be delegating IO through a *striped* processor, using 2 stripes for the object and a stripe width of 512K. The dispatcher in turn instantiates 8 processors (one for each stripe), each configured with the information stored in the object associated with each stripe (e.g., stripe 0 uses object 103). Finally, when the stripe dispatcher performs IO on stripe 0, it will use the context in the object descriptor for object 103 to instantiate a *replicated* processor, which mirrors writes to the NADs listed in its replica set, and issues reads to the nearest in sync replica (where distance is currently simply local or remote).

In addition to the striping and mirroring processors described here, the map can support other more advanced processors, such as erasure coding, or byte-range mappings to arbitrary objects (which supports among other things data deduplication).

3.2 Dispatch

IO requests are handled by a chain of dispatchers, each of which has some common functionality. Dispatchers may have to fragment requests into pieces if they span the ranges covered by different subprocessors, or clone requests into multiple subrequests (e.g., for replication), and they must collect the results of subrequests and deal with partial failures.

The replication and striping modules included in the standard library are representative of the ways processors transform requests as they traverse a dispatch stack. The replication processor allows a request to be split and issued concurrently to a set of replica objects. The request address remains unchanged within each object, and responses are not returned until all replicas have acknowledged a request as complete. The processor prioritizes reading from local replicas, but forwards requests to remote replicas in the event of a failure (either an error response or a timeout). It imposes a global ordering on write requests and streams them to all replicas in parallel. It also periodically commits a light-weight checkpoint to each replica's log to maintain a persistent record of synchronization points; these checkpoints are used for crash recovery (Section 5.1.3).

The striping processor distributes data across a collection of sparse objects. It is parameterized to take a stripe size (in bytes) and a list of objects to act as the ordered stripe set. In the event that a request crosses a stripe boundary, the processor splits that request into a set of per-stripe requests and issues those asynchronously, collecting the responses before returning. Static, address-based striping is a relatively simple load balancing and data distribution mechanism as compared to placement schemes such as consistent hashing [20]. Our experience has been that the approach is effective, because data placement tends to be reasonably uniform within an object address space, and because using a reasonably large stripe size (we default to 512KB) preserves locality well enough to keep request fragmentation overhead low in normal operation.

3.3 Coherence

Strata clients also participate in a simple coordination protocol in order to allow the virtual address map for a virtual object to be updated even while that object is in use. Online reconfiguration provides a means for recovering from failures, responding to capacity changes, and even moving objects in response to observed or predicted load (on a device basis — this is distinct from client load balancing, which we also support through a switch-based protocol described in Section 6.2).

The virtual address maps are stored in a distributed, synchronized *configuration database* implemented over Apache Zookeeper, which is also available for any low-bandwidth synchronization required by services elsewhere in the software stack. The coherence protocol is built on top of the configuration database. It is currently optimized for a single writer per object, and works as follows: when a client wishes to write to a virtual object, it first claims a lock for it in the configuration database. If the object is already locked, the client requests that the holder release it so that the client can claim it. If the holder does not voluntarily release it within a reasonable time, the holder is considered unresponsive and fenced from the system using the mechanism described in Section 6.2. This is enough to allow movement of objects, by first creating new, out of sync physical objects at the desired location, then requesting a release of the object's lock holder if there is one. The user of the object will reacquire the lock on the next write, and in the process discover the new out of sync replica and initiate resynchronization. When the new replica is in sync, the same process may be repeated to delete replicas that are at undesirable locations.

4 Network Attached Disks

The unit of storage in Strata is a Network Attached Disk (NAD), consisting of a balanced combination of CPU, network and storage components. In our current hardware, each NAD has two 10 gigabit Ethernet ports, two PCIe flash cards capable of 10 gigabits of throughput each, and a pair of Xeon processors that can keep up with request load and host additional services alongside the data path. Each NAD provides two distinct services.

First, it efficiently multiplexes the raw storage hardware across multiple concurrent users, using an object storage protocol. Second, it hosts applications that provide higher level services over the cluster. Object rebalancing (Section 5.2.1) and the NFS protocol interface (Section 6.1) are examples of these services.

At the device level, we multiplex the underlying storage into objects, named by 128-bit identifiers and consisting of sparse 2^{64} byte data address spaces. These address spaces are currently backed by a garbage-collected log-structured object store, but the implementation of the object store is opaque to the layers above and could be replaced if newer storage technologies made different access patterns more efficient. We also provide increased capacity by allowing each object to flush low priority or infrequently used data to disk, but this is again hidden behind the object interface. The details of disk tiering, garbage collection, and the layout of the file system are beyond the scope of this paper.

The physical object interface is for the most part a traditional object-based storage device [37,38] with a CRUD interface for sparse objects, as well as a few extensions to assist with our clustering protocol (Section 5.1.2). It is significantly simpler than existing block device interfaces, such as the SCSI command set, but is also intended to be more direct and general purpose than even narrower interfaces such as those of a key-value store. Providing a low-level hardware abstraction layer allows the implementation to be customized to accommodate best practices of individual flash implementations, and also allows more dramatic design changes at the media interface level as new technologies become available.

4.1 Network Integration

As with any distributed system, we must deal with misbehaving nodes. We address this problem by tightly coupling with managed Ethernet switches, which we discuss at more length in Section 6.2. This approach borrows ideas from systems such as Sane [8] and Ethane [7], in which a managed network is used to enforce isolation between independent endpoints. The system integrates with both OpenFlow-based switches and software switching at the VMM to ensure that Strata objects are only addressable by their authorized clients.

Our initial implementation used Ethernet VLANs, because this form of hardware-supported isolation is in common use in enterprise environments. In the current implementation, we have moved to OpenFlow, which provides a more flexible tunneling abstraction for traffic isolation.

We also expose an isolated private virtual network for out-of-band control and management operations internal to the cluster. This allows NADs themselves to access remote objects for peer-wise resynchronization and reorganization under the control of a cluster monitor.

5 Online Reconfiguration

There are two broad categories of events to which Strata must respond in order to maintain its performance and reliability properties. The first category includes faults that occur directly on the data path. The dispatch library recovers from such faults immediately and automatically by reconfiguring the affected virtual objects on behalf of the client. The second category includes events such as device failures and load imbalance. These are handled by a dedicated *cluster monitor* which performs large-scale reconfiguration tasks to maintain the health of the system as a whole. In all cases, reconfiguration is performed online and has minimal impact on client availability.

5.1 Object Reconfiguration

A number of error recovery mechanisms are built directly into the dispatch library. These mechanisms allow clients to quickly recover from failures by reconfiguring individual virtual objects on the data path.

5.1.1 IO Errors

The replication IO processor responds to read errors in the obvious way: by immediately resubmitting failed requests to different replicas. In addition, clients maintain per-device error counts; if the aggregated error count for a device exceeds a configurable threshold, a background task takes the device offline and coordinates a system-wide reconfiguration (Section 5.2.2).

IO processors respond to write errors by synchronously reconfiguring virtual objects at the time of the failure. This involves three steps. First, the affected replica is marked *out of sync* in the configuration database. This serves as a global, persistent indication that the replica may not be used to serve reads because it contains potentially stale data. Second, a best-effort attempt is made to inform the NAD of the error so that it can initiate a background task to resynchronize the affected replica. This allows the system to recover from transient failures almost immediately. Finally, the IO processor allocates a special *patch* object on a separate device and adds this to the replica set. Once a replica has been marked out of sync, no further writes are issued to it until it has been resynchronized; patches prevent device failures from impeding progress by providing a temporary buffer to absorb writes under these degraded conditions. With the patch object allocated, the IO processor can continue to meet the replication requirements for new writes while out of sync replicas are repaired in the background. A replica set remains available as long as an in sync replica or an out of sync replica *and* all of its patches are available.

5.1.2 Resynchronization

In addition to providing clients direct access to devices via virtual address maps, Strata provides a number of background services to maintain the health of individual virtual objects and the system as a whole. The most fundamental of these is the *resync* service, which provides a background task that can resynchronize objects replicated across multiple devices.

Resync is built on top of a special NAD `resync` API that exposes the underlying log structure of the object stores. NADs maintain a Log Serial Number (LSN) with every physical object in their stores; when a record is appended to an object's log, its LSN is monotonically incremented. The IO processor uses these LSNs to impose a global ordering on the changes made to physical objects that are replicated across stores and to verify that all replicas have received all updates.

If a write failure causes a replica to go out of sync, the client can request the system to resynchronize the replica. It does this by invoking the `resync` RPC on the NAD which hosts the out of sync replica. The server then starts a background task which streams the missing log records from an in sync replica and applies them to the local out of sync copy, using the LSN to identify which records the local copy is missing.

During resync, the background task has exclusive write access to the out of sync replica because all clients have been reconfigured to use patches. Thus the resync task can chase the tail of the in sync object's log while clients continue to write. When the bulk of the data has been copied, the resync task enters a final *stop-and-copy* phase in which it acquires exclusive write access to all replicas in the replica set, finalizes the resync, applies any client writes received in the interim, marks the replica as in sync in the configuration database, and removes the patch.

It is important to ensure that resync makes timely progress to limit vulnerability to data loss. Very heavy client write loads may interfere with resync tasks and, in the worst case, result in unbounded transfer times. For this reason, when an object is under resync, client writes are throttled and resync requests are prioritized.

5.1.3 Crash Recovery

Special care must be taken in the event of an unclean shutdown. On a clean shutdown, all objects are released by removing their locks from the configuration database. Crashes are detected when replica sets are discovered with stale locks (i.e., locks identifying unresponsive IO processors). When this happens, it is not safe to assume that replicas marked *in sync* in the configuration database are truly in sync, because a crash might have occured midway through a the configuration database update; instead, all the replicas in the set must be queried directly to determine their states.

In the common case, the IO processor retrieves the LSN for every replica in the set and determines which replicas, if any, are out of sync. If all replicas have the same LSN, then no resynchronization is required. If different LSNs are discovered, then the replica with the highest LSN is designated as the authoritative copy, and all other replicas are marked out of sync and resync tasks are initiated.

If a replica cannot be queried during the recovery procedure, it is marked as *diverged* in the configuration database and the replica with the highest LSN from the remaining available replicas is chosen as the authoritative copy. In this case, writes may have been committed to the diverged replica that were not committed to any others. If the diverged replica becomes available again some time in the future, these extra writes must be discarded. This is achieved by rolling the replica back to its last checkpoint and starting a resync from that point in its log. Consistency in the face of such rollbacks is guaranteed by ensuring that objects are successfully marked out of sync in the configuration database *before* writes are acknowledged to clients. Thus write failures are guaranteed to either mark replicas out of sync in the configuration database (and create corresponding patches) or propagate back to the client.

5.2 System Reconfiguration

Strata also provides a highly-available monitoring service that watches over the health of the system and coordinates system-wide recovery procedures as necessary. Monitors collect information from clients, SMART diagnostic tools, and NAD RPCs to gauge the status of the system. Monitors build on the per-object reconfiguration mechanisms described above to respond to events that individual clients don't address, such as load imbalance across the system, stores nearing capacity, and device failures.

5.2.1 Rebalance

Strata provides a rebalance facility which is capable of performing system-wide reconfiguration to repair broken replicas, prevent NADs from filling to capacity, and improve load distribution across NADs. This facility is in turn used to recover from device failures and expand onto new hardware.

Rebalance proceeds in two stages. In the first stage, the monitor retrieves the current system configuration, including the status of all NADs and virtual address map of every virtual object. It then constructs a new layout for the replicas according to a customizable placement policy. This process is scriptable and can be easily tailored to suit specific performance and durability requirements for individual deployments (see Section 7.3 for some analysis of the effects of different placement policies). The default policy uses a greedy algorithm that considers a number of criteria designed to ensure that replicated physical objects do not share fault domains, capacity imbalances are avoided as much as possible, and migration overheads are kept reasonably low. The new layout is formulated as a rebalance plan describing what changes need to be applied to individual replica sets to achieve the desired configuration.

In the second stage, the monitor coordinates the execution of the rebalance plan by initiating resync tasks on individual NADs to effect the necessary data migration. When replicas need to be moved, the migration is performed in three steps:

1. A new replica is added to the destination NAD
2. A resync task is performed to transfer the data
3. The old replica is removed from the source NAD

This requires two reconfiguration events for the replica set, the first to extend it to include the new replica, and the second to prune the original after the resync has completed. The monitor coordinates this procedure across all NADs and clients for all modified virtual objects.

5.2.2 Device Failure

Strata determines that a NAD has failed either when it receives a hardware failure notification from a responsive NAD (such as a failed flash device or excessive error count) or when it observes that a NAD has stopped responding to requests for more than a configurable timeout. In either case, the monitor responds by taking the NAD offline and initiating a system-wide reconfiguration to repair redundancy.

The first thing the monitor does when taking a NAD offline is to disconnect it from the data path VLAN. This is a strong benefit of integrating directly against an Ethernet switch in our environment: prior to taking corrective action, the NAD is synchronously disconnected from the network for all request traffic, avoiding the distributed systems complexities that stem from things such as overloaded components appearing to fail and then returning long after a timeout in an inconsistent state. Rather than attempting to use completely end-host mechanisms such as watchdogs to trigger reboots, or agreement protocols to inform all clients of a NAD's failure, Strata disables the VLAN and requires that the failed NAD reconnect on the (separate) control VLAN in the event that it returns to life in the future.

From this point, the recovery logic is straight forward. The NAD is marked as failed in the configuration database and a rebalance job is initiated to repair any replica sets containing replicas on the failed NAD.

5.2.3 Elastic Scale Out

Strata responds to the introduction of new hardware much in the same way that it responds to failures. When the monitor observes that new hardware has been installed, it uses the rebalance facility to generate a layout that incorporates the new devices. Because replication is generally configured underneath striping, we can migrate virtual objects at the granularity of individual stripes, allowing a single striped file to exploit the aggregated performance of many devices. Objects, whether whole files or individual stripes, can be moved to another NAD even while the file is online, using the existing resync mechanism. New NADs are populated in a controlled manner to limit the impact of background IO on active client workloads.

6 Storage Protocols

Strata supports legacy protocols by providing an execution runtime for hosting protocol servers. Protocols are built as thin presentation layers on top of the dispatch interfaces; multiple protocol instances can operate side by side. Implementations can also leverage SDN-based protocol scaling to transparently spread multiple clients across the distributed runtime environment.

6.1 Scalable NFS

Strata is designed so that application developers can focus primarily on implementing protocol specifications without worrying much about how to organize data on disk. We expect that many storage protocols can be implemented as thin wrappers around the provided dispatch library. Our NFS implementation, for example, maps very cleanly onto the high-level dispatch APIs, providing only protocol-specific extensions like RPC marshalling and NFS-style access control. It takes advantage of the configuration database to store mappings between the NFS namespace and the backend objects, and it relies exclusively on the striping and replication processors to implement the data path. Moreover, Strata allows NFS servers to be instantiated across multiple backend nodes, automatically distributing the additional processing overhead across backend compute resources.

6.2 SDN Protocol Scaling

Scaling legacy storage protocols can be challenging, especially when the protocols were not originally designed for a distributed back end. Protocol scalability limitations may not pose significant problems for traditional arrays, which already sit behind relatively narrow network interfaces, but they can become a performance bottleneck in Strata's distributed architecture.

A core property that limits scale of access bandwidth of conventional IP storage protocols is the presentation of storage servers behind a single IP address. Fortunately, emerging "software defined" network (SDN) switches provide interfaces that allow applications to take more precise control over packet forwarding through Ethernet switches than has traditionally been possible.

Using the OpenFlow protocol, a software controller is able to interact with the switch by pushing flow-specific rules onto the switch's forwarding path. OpenFlow rules are effectively wild-carded packet filters and associated actions that tell a switch what to do when a matching packet is identified. SDN switches (our implementation currently uses an Arista Networks 7050T-52) interpret these flow rules and push them down onto the switch's TCAM or L2/L3 forwarding tables.

By manipulating traffic through the switch at the granularity of individual flows, Strata protocol implementations are able to present a single logical IP address to multiple clients. Rules are installed on the switch to trigger a fault event whenever a new NFS session is opened, and the resulting exception path determines which protocol instance to forward that session to initially. A service monitors network activity and migrates client connections as necessary to maintain an even workload distribution.

The protocol scaling API wraps and extends the conventional socket API, allowing a protocol implementation to bind to and listen on a shared IP address across all of its instances. The client load balancer then monitors the traffic demands across all of these connections and initiates flow migration in response to overload on any individual physical connection.

In its simplest form, client migration is handled entirely at the transport layer. When the protocol load balancer observes that a specific NAD is overloaded, it updates the routing tables to redirect the busiest client workload to a different NAD. Once the client's traffic is diverted, it receives a TCP `RST` from the new NAD and establishes a new connection, thereby transparently migrating traffic to the new NAD.

Strata also provides hooks for situations where application layer coordination is required to make migration safe. For example, our NFS implementation registers a pre-migration routine with the load balancer, which allows the source NFS server to flush any pending, non-idempotent requests (such as `create` or `remove`) before the connection is redirected to the destination server.

7 Evaluation

In this section we evaluate our system both in terms of effective use of flash resources, and as a scalable, reliable provider of storage for NFS clients. First, we establish baseline performance over a traditional NFS server on the same hardware. Then we evaluate how performance scales as nodes are added and removed from the system, using VM-based workloads over the legacy NFS interface, which is oblivious to cluster changes. In addition, we compare the effects of load balancing and object placement policy on performance. We then test reliability in the face of node failure, which is a crucial feature of any distributed storage system. We also examine the relation between CPU power and performance in our system as a demonstration of the need to balance node power between flash, network and CPU.

7.1 Test environment

Evaluation was performed on a cluster of the maximum size allowed by our 48-port switch: 12 NADs, each of which has two 10 gigabit Ethernet ports, two 800 GB Intel 910 PCIe flash cards, 6 3 TB SATA drives, 64 GB of RAM, and 2 Xen E5-2620 processors at 2 GHz with 6 cores/12 threads each, and 12 clients, in the form of Dell PowerEdge R420 servers running ESXi 5.0, with two 10 gigabit ports each, 64 GB of RAM, and 2 Xeon E5-2470 processors at 2.3 GHz with 8 cores/16 threads each. We configured the deployment to maintain two replicas of every stored object, without striping (since it unnecessarily complicates placement comparisons and has little benefit for symmetric workloads). Garbage collection is active, and the deployment is in its standard configuration with a disk tier enabled, but the workloads have been configured to fit entirely within flash, as the effects of

Server	Read IOPS	Write IOPS
Strata	40287	9960
KNFS	23377	5796

Table 1: Random IO performance on Strata versus KNFS.

cache misses to magnetic media are not relevant to this paper.

7.2 Baseline performance

To provide some performance context for our architecture versus a typical NFS implementation, we compare two minimal deployments of NFS over flash. We set Strata to serve a single flash card, with no replication or striping, and mounted it loopback. We ran a fio [34] workload with a 4K IO size 80/20 read-write mix at a queue depth of 128 against a fully allocated file. We then formatted the flash card with ext4, exported it with the linux kernel NFS server, and ran the same test. The results are in Table 1. As the table shows, we offer good NFS performance at the level of individual devices. In the following section we proceed to evaluate scalability.

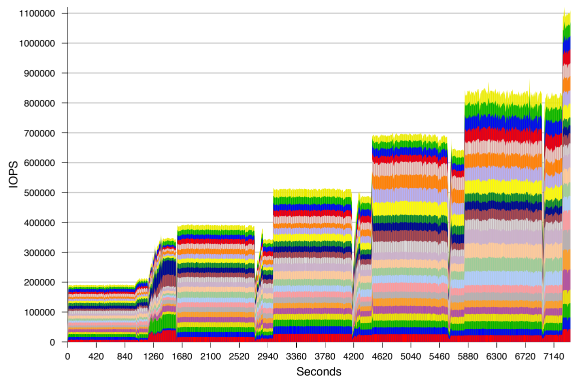


Figure 4: IOPS over time, read-only workload.

7.3 Scalability

In this section we evaluate how well performance scales as we add NADs to the cluster. We begin each test by deploying 96 VMs (8 per client) into a cluster of 2 NADs. We choose this number of VMs because ESXi limits the queue depth for a VM to 32 outstanding requests, but we do not see maximum performance until a queue depth of 128 per flash card. The VMs are each configured to run the same fio workload for a given test. In Figure 4, fio generates 4K random reads to focus on IOPS scalability. In Figure 5, fio generates an 80/20 mix of reads and writes at 128K block size in a Pareto distribution such that 80% of requests go to 20% of the data. This is meant to be more representative of real VM workloads, but with enough offered load to completely saturate the cluster.

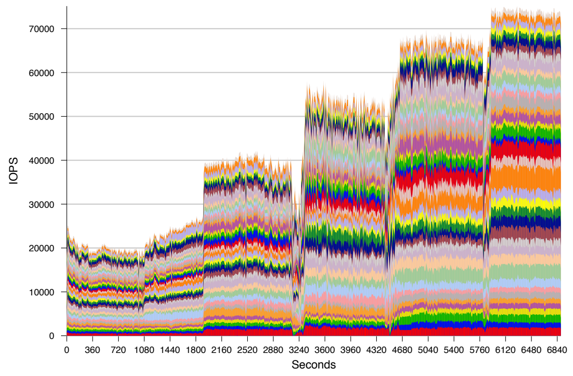


Figure 5: IOPS over time, 80/20 R/W workload.

As the tests run, we periodically add NADs, two at a time, up to a maximum of twelve[2]. When each pair of NADs comes online, a rebalancing process automatically begins to move data across the cluster so that the amount of data on each NAD is balanced. When it completes, we run in a steady state for two minutes and then add the next pair. In both figures, the periods where rebalancing is in progress are reflected by a temporary drop in performance (as the rebalance process competes with client workloads for resources), followed by a rapid increase in overall performance when the new nodes are marked available, triggering the switch to load-balance clients to them. A cluster of 12 NADs achieves over 1 million IOPS in the IOPS test, and 10 NADs achieve 70,000 IOPS (representing more than 9 gigabytes/second of throughput) in the 80/20 test.

We also test the effect of placement and load balancing on overall performance. If the location of a workload source is unpredictable (as in a VM data center with virtual machine migration enabled), we need to be able to migrate clients quickly in response to load. However, if the configuration is more static or can be predicted in advance, we may benefit from attempting to place clients and data together to reduce the network overhead incurred by remote IO requests. As discussed in Section 5.2.1, the load-balancing and data migration features of Strata make both approaches possible. Figure 4 is the result of an aggressive local placement policy, in which data is placed on the same NAD as its clients, and both are moved as the number of devices changes. This achieves the best possible performance at the cost of considerable data movement. In contrast, Figure 6 shows the

[2]ten for the read/write test due to an unfortunate test harness problem

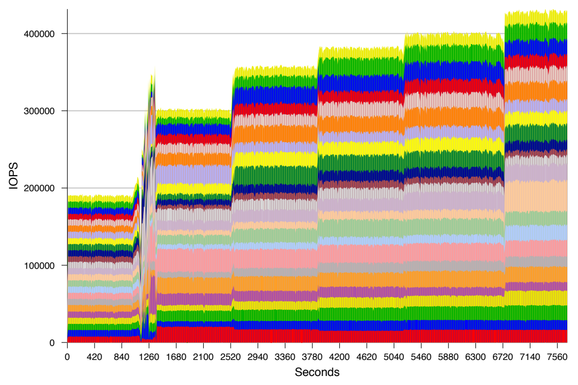


Figure 6: IOPS over time, read-only workload with random placement

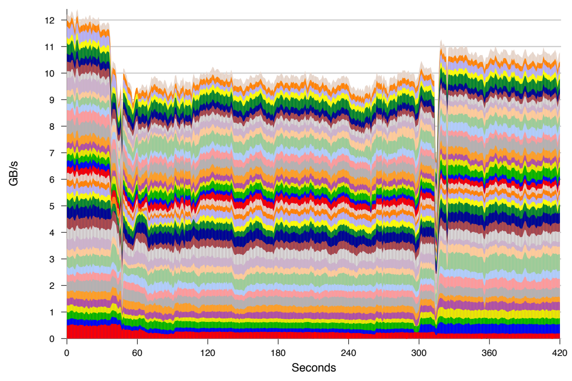


Figure 7: Aggregate bandwidth for 80/20 clients during failover and recovery

performance of an otherwise identical test configuration when data is placed randomly (while still satisfying fault tolerance and even distribution constraints), rather than being moved according to client requests. The pareto workload (Figure 5) is also configured with the default random placement policy, which is the main reason that it does not scale linearly: as the number of nodes increases, so does the probability that a request will need to be forwarded to a remote NAD.

7.4 Node Failure

As a counterpoint to the scalability tests run in the previous section, we also tested the behaviour of the cluster when a node is lost. We configured a 10 NAD cluster with 10 clients hosting 4 VMs each, running the 80/20 Pareto workload described earlier. Figure 7 shows the behaviour of the system during this experiment. After the VMs had been running for a short time, we powered off one of the NADs by IPMI, waited 60 seconds, then powered it back on. During the node outage, the system continued to run uninterrupted but with lower throughput. When the node came back up, it spent some time resynchronizing its objects to restore full replication to the system, and then rejoined the cluster. The client load balancer shifted clients onto it and throughput was restored (within the variance resulting from the client load balancer's placement decisions).

7.5 Protocol overhead

The benchmarks up to this point have all been run inside VMs whose storage is provided by a virtual disk that Strata exports by NFS to ESXi. This configuration requires no changes on the part of the clients to scale across a cluster, but does impose overheads. To quantify these overheads we wrote a custom fio engine that

CPU	IOPS	Freq (Cores)	Price
E5-2620	127K	2 GHz (6)	$406
E5-2640	153K (+*20%*)	2.5 GHz (6)	$885
E5-2650v2	188K (+*48%*)	2.6 GHz (8)	$1166
E5-2660v2	183K (+*44%*)	2.2 GHz (10)	$1389

Table 2: Achieved IOPS on an 80/20 random 4K workload across 2 MicroArrays

is capable of performing IO directly against our native dispatch interface (that is, the API by which our NFS protocol gateway interacts with the NADs). We then compared the performance of a single VM running a random 4k read fio workload (for maximum possible IOPS) against a VMDK exported by NFS to the same workload run against our native dispatch engine. In this experiment, the VMDK-based experiment produced an average of 50240 IOPS, whereas direct access achieved 54060 IOPS, for an improvement of roughly 8%.

7.6 Effect of CPU on Performance

A workload running at full throttle with small requests completely saturates the CPU. This remains true despite significant development effort in performance debugging, and a great many improvements to minimize data movement and contention. In this section we report the performance improvements resulting from faster CPUs. These results are from random 4K NFS requests in an 80/20 readwrite mix at 128 queue depth over four 10Gb links to a cluster of two NADs, each equipped with 2 physical CPUs.

Table 2 shows the results of these tests. In short, it is possible to "buy" additional storage performance under full load by upgrading the CPUs into a more "balanced" configuration. The wins are significant and carry a non-trivial increase in the system cost. As a result of this experimentation, we elected to use a higher performance CPU in the shipping version of the product.

8 Related Work

Strata applies principles from prior work in server virtualization, both in the form of hypervisor [5,32] and lib-OS [14] architectures, to solve the problem of sharing and scaling access to fast non-volatile memories among a heterogeneous set of clients. Our contributions build upon the efforts of existing research in several areas.

Recently, researchers have begin to investigate a broad range of system performance problems posed by storage class memory in single servers [3], including current PCIe flash devices [30], next generation PCM [1], and byte addressability [13]. Moneta [9] proposed solutions to an extensive set of performance bottlenecks over the PCIe bus interface to storage, and others have investigated improving the performance of storage class memory through polling [33], and avoiding system call overheads altogether [10]. We draw from this body of work to optimize the performance of our dispatch library, and use this baseline to deliver a high performance scale-out network storage service. In many cases, we would benefit further from these efforts—for example, our implementation could be optimized to offload per-object access control checks, as in Moneta-D [10]. There is also a body of work on efficiently using flash as a caching layer for slower, cheaper storage in the context of large file hosting. For example, S-CAVE [23] optimizes cache utilization on flash for multiple virtual machines on a single VMware host by running as a hypervisor module. This work is largely complementary to ours; we support using flash as a caching layer and would benefit from more effective cache management strategies.

Prior research into scale-out storage systems, such as FAWN [2], and Corfu [4] has considered the impact of a range of NV memory devices on cluster storage performance. However, to date these systems have been designed towards lightweight processors paired with simple flash devices. It is not clear that this balance is the correct one, as evidenced by the tendency to evaluate these same designs on significantly more powerful hardware platforms than they are intended to operate [4]. Strata is explicitly designed for dense virtualized server clusters backed by performance-dense PCIe-based non-volatile memory. In addition, like older commodity disk-oriented systems including Petal [22,29] and FAB [28], prior storage systems have tended to focus on building aggregation features at the lowest level of their designs, and then adding a single presentation layer on top. Strata in contrasts isolates shares each powerful PCIe-based storage class memory as its underlying primitive. This has allowed us to present a scalable runtime environment in which multiple protocols can coexist as peers without sacrificing the raw performance that today's high performance memory can provide. Many scale-out storage systems, including NV-Heaps [12], Ceph/RADOS [31], and even PNFS [18] are unable to support the legacy formats in enterprise environments. Our agnosticism to any particular protocol is similar to approach used by Ursa Minor [16], which also boasted a versatile client library protocol to share access to a cluster of magnetic disks.

Strata does not attempt to provide storage for datacenter-scale environments, unlike systems including Azure [6], FDS [26], or Bigtable [11]. Storage systems in this space differ significantly in their intended workload, as they emphasize high throughput linear operations. Strata's managed network would also need to be extended to support datacenter-sized scale out. We also differ from in-RAM approaches such a RAMCloud [27] and memcached [15], which offer a different class of durability guarantee and cost.

9 Conclusion

Storage system design faces a sea change resulting from the dramatic increase in the performance density of its component media. Distributed storage systems composed of even a small number of network-attached flash devices are now capable of matching the offered load of traditional systems that would have required multiple racks of spinning disks.

Strata is an enterprise storage architecture that responds to the performance characteristics of PCIe storage devices. Using building blocks of well-balanced flash, compute, and network resources and then pairing the design with the integration of SDN-based Ethernet switches, Strata provides an incrementally deployable, dynamically scalable storage system.

Strata's initial design is specifically targeted at enterprise deployments of VMware ESX, which is one of the dominant drivers of new storage deployments in enterprise environments today. The system achieves high performance and scalability for this specific NFS environment while allowing applications to interact directly with virtualized, network-attached flash hardware over new protocols. This is achieved by cleanly partitioning our storage implementation into an underlying, low-overhead virtualization layer and a scalable framework for implementing storage protocols. Over the next year, we intend to extend the system to provide general-purpose NFS support by layering a scalable and distributed metadata service and small object support above the base layer of coarse-grained storage primitives.

References

[1] Akel, A., Caulfield, A. M., Mollov, T. I., Gupta, R. K., and Swanson, S. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems* (Berkeley, CA, USA, 2011), HotStorage'11, USENIX Association, pp. 2–2.

[2] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), SOSP '09, pp. 1–14.

[3] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 2–2.

[4] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), NSDI'12.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), SOSP '03, pp. 164–177.

[6] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, pp. 143–157.

[7] Casado, M., Freedman, M. J., Pettit, J., Luo, J., Mckeown, N., and Shenker, S. Ethane: Taking control of the enterprise. In *In SIGCOMM Computer Comm. Rev* (2007).

[8] Casado, M., Garfinkel, T., Akella, A., Freedman, M. J., Boneh, D., McKeown, N., and Shenker, S. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[9] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), MICRO '43, pp. 385–395.

[10] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 387–400.

[11] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[12] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[13] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 133–146.

[14] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (1995), SOSP '95, pp. 251–266.

[15] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[16] GANGER, G. R., ABD-EL-MALEK, M., CRANOR, C., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., AND WYLIE, J. J. Ursa minor: versatile cluster-based storage, 2005.

[17] GIBSON, G. A., AMIRI, K., AND NAGLE, D. F. A case for network-attached secure disks. Tech. Rep. CMU-CS-96-142, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), Pittsburgh, 1996.

[18] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *IN PROCEEDINGS OF 22ND IEEE/13TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST* (2005).

[19] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng. 17*, 1 (Jan. 1991), 64–76.

[20] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), STOC '97, pp. 654–663.

[21] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[22] LEE, E. K., AND THEKKATH, C. A. Petal: distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems* (1996), ASPLOS VII, pp. 84–92.

[23] LUO, T., MA, S., LEE, R., ZHANG, X., LIU, D., AND ZHOU, L. S-cave: Effective ssd caching to improve virtual machine storage performance. In *Parallel Architectures and Compilation Techniques* (2013), PACT '13, pp. 103–112.

[24] MEYER, D. T., CULLY, B., WIRES, J., HUTCHINSON, N. C., AND WARFIELD, A. Block mason. In *Proceedings of the First conference on I/O virtualization* (2008), WIOV'08.

[25] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation* (1996), OSDI '96, pp. 153–167.

[26] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 1–15.

[27] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[28] SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ASPLOS XI, ACM, pp. 48–58.

[29] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: a scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (1997), SOSP '97, pp. 224–237.

[30] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 8:1–8:13.

[31] WEIL, S. A., WANG, F., XIN, Q., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable object-based storage system. Tech. rep., 2006.

[32] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop* (2002).

[33] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 3–3.

[34] Flexible io tester. http://git.kernel.dk/?p=fio.git;a=summary.

[35] Linux device mapper resource page. http://sourceware.org/dm/.

[36] Linux logical volume manager (lvm2) resource page. http://sourceware.org/lvm2/.

[37] Seagate kinetic open storage documentation. https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki.

[38] Scsi object-based storage device commands - 2, 2011. http://www.incits.org/scopes/1729.htm.