Treating the Storage Stack Like a Network

IOAN STEFANOVICI, Microsoft Research
BIANCA SCHROEDER, University of Toronto
GREG O'SHEA, Microsoft Research
ENO THERESKA, Confluent, Imperial College London

2

In a data center, an IO from an application to distributed storage traverses not only the network but also several software stages with diverse functionality. This set of ordered stages is known as the storage or IO stack. Stages include caches, hypervisors, IO schedulers, file systems, and device drivers. Indeed, in a typical data center, the number of these stages is often larger than the number of network hops to the destination. Yet, while packet routing is fundamental to networks, no notion of IO routing exists on the storage stack. The path of an IO to an endpoint is predetermined and hard coded. This forces IO with different needs (e.g., requiring different caching or replica selection) to flow through a one-size-fits-all IO stack structure, resulting in an ossified IO stack.

This article proposes sRoute, an architecture that provides a routing abstraction for the storage stack. sRoute comprises a centralized control plane and "sSwitches" on the data plane. The control plane sets the forwarding rules in each sSwitch to route IO requests at runtime based on application-specific policies. A key strength of our architecture is that it works with unmodified applications and Virtual Machines (VMs). This article shows significant benefits of customized IO routing to data center tenants: for example, a factor of 10 for tail IO latency, more than 60% better throughput for a customized replication protocol, a factor of 2 in throughput for customized caching, and enabling live performance debugging in a running system.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Data centers, storage, software-defined storage, SDS, routing, storage stack

ACM Reference Format:
Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. 2017. Treating the storage stack like a network. ACM Trans. Storage 13, 1, Article 2 (February 2017), 27 pages.
DOI: http://dx.doi.org/10.1145/3032968

1. INTRODUCTION

An application's IO stack is rich in stages providing compute, network, and storage functionality. These stages include guest OSes, file systems, hypervisors, network appliances, and distributed storage with caches and schedulers. Indeed, there are over 18+ types of stages on a typical data center IO stack [Thereska et al. 2013]. Furthermore, most IO stacks support the injection of new stages with new functionality using

Authors' addresses: I. Stefanovici and G. O'Shea, Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, United Kingdom; emails: {t-iostef, gregos}@microsoft.com; B. Schroeder, Department of Computer Science, University of Toronto, 10 King's College Road, Rm.3302, Toronto, Ontario, M5S 3G4, Canada; email: bianca@cs.toronto.edu; E. Thereska, Confluent, Inc., 101 University Ave, Suite 111, Palo Alto, CA, 94301, United States; email: eno.thereska@gmail.com.

DOI: http://dx.doi.org/10.1145/3032968

filter drivers common in most OSes [Microsoft Corporation 2014b; FreeBSD 2014; Love 2010] or appliances over the network [Sherry et al. 2012].

Controlling or programming how IOs flow through this stack is hard, if not impossible, for tenants and service providers alike. Once an IO enters the system, the path to its endpoint is pre-determined and static. It must pass through all stages on the way to the endpoint. A new stage with new functionality means a longer path with added latency for every IO. As raw storage and networking speeds improve, the length of the IO stack is increasingly becoming a new bottleneck [Peter et al. 2014]. Furthermore, the IO stack stages have narrow interfaces and operate in isolation. Unlocking functionality often requires coordinating the functionality of multiple such stages. These reasons lead to applications running on a general-purpose IO stack that cannot be tuned to any of their specific needs or to one-off customized implementations that require application and system rewrite.

This article's main contribution is experimenting with applying a well-known networking primitive, *routing*, to the storage stack. IO routing provides the ability to dynamically change the path and destination of an IO, like a `read` or `write`, at runtime. Control plane applications use IO routing to provide customized data plane functionality for tenants and data center services.

Consider three specific examples of how routing is useful. In one example, a load balancing service selectively routes `write` requests to go to less-loaded servers, while ensuring `read` requests are always routed to the latest version of the data (Section 5.1). In another example, a control application provides per-tenant throughput versus latency tradeoffs for replication update propagation by using IO routing to set a tenant's IO read- and write-set at runtime (Section 5.2). In a third example, a control application can route requests to per-tenant caches to maintain cache isolation (Section 5.3). Finally, a control application enables dynamic performance debugging and bottleneck isolation in data paths that span several machines (Section 5.4).

IO routing is challenging because the storage stack is stateful. Routing a `write` IO through one path to endpoint A and a subsequent `read` IO through a different path or to a different endpoint B needs to be mindful of application consistency needs. Another key challenge is data plane efficiency. Changing the path of an IO at runtime requires determining where on the data plane to insert storage switches to minimize the number of times an IO traverses them, as well as to minimize IO processing times.

We designed and implemented sRoute, a system that enables IO routing in the storage stack. sRoute's approach builds on the IOFlow storage architecture [Thereska et al. 2013]. IOFlow already provides a separate control plane for storage traffic and a logically centralized controller with global visibility over the data center topology. As an analogy to networking, sRoute builds on IOFlow just like software-defined networking (SDN) functions build on OpenFlow [McKeown et al. 2008]. IOFlow also made a case for request routing. However, it only explored the concept of *bypassing* stages along the IO path and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This article provides a more complete routing abstraction.

This article makes the following contributions:

—We propose an IO routing abstraction for the IO stack.
—sRoute provides per-IO and per-flow routing configuration updates with strong semantic guarantees.
—sRoute provides an efficient control plane. It does so by distributing the control plane logic required for IO routing using *delegate functions*.
—We report on building four control applications using IO routing: tail latency control, replica set control, file caching control, and performance debugging.

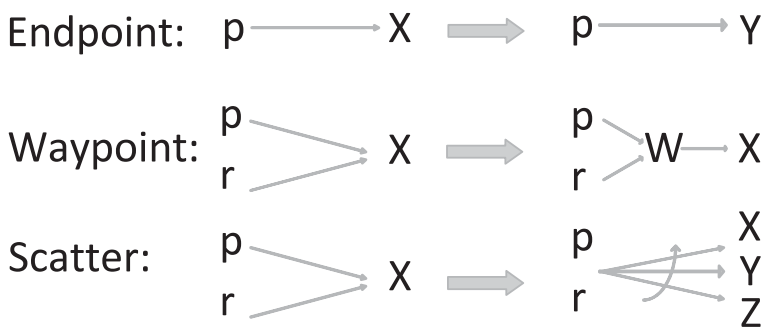

Fig. 1. Three types of IO routing: endpoint, waypoint, and scatter. p, r refer to sources such as VMs or containers. X, Y, Z are endpoints such as files. W represents a waypoint stage with specialized functionality, for example, a file cache or scheduler.

—We provide a detailed road map for future work on software-defined storage that we hope will encourage further research in the area.

The results of our evaluation demonstrate that data center tenants benefit significantly from IO stack customization. The benefits can be provided to today's unmodified tenant applications and VMs. Furthermore, writing specialized control applications is straightforward because they use a common IO routing abstraction.

2. ROUTING TYPES AND CHALLENGES

The data plane, or *IO stack*, comprises all the stages an IO request traverses from an application until it reaches its destination. For example, a read to a file will traverse a guest OS's file system, buffer cache, scheduler, and then similar stages in the hypervisor, followed by OSes, file systems, caches, and device drivers on remote storage servers. We define per-IO routing in this context as the ability to control the IO's endpoint as well as the path to that endpoint. The first question is what the above definition means for storage semantics. A second question is whether IO routing is a useful abstraction.

To address the first question, we looked at a large set of storage system functionalities and distilled from them three types of IO routing that make sense semantically in the storage stack. Figure 1 illustrates these three types. In *endpoint* routing, IO from a source p to a destination file X is routed to another destination file Y. In *waypoint* routing, IOs from sources p and r to a file X are first routed to a specialized stage W. In *scatter* routing, IOs from p and r are routed to a subset of data replicas.

This article makes the case that IO routing is a useful abstraction. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. Table I shows a diverse set of such storage stack functionalities, categorized according to the type of IO routing that enables them.

Endpoint routing. Routes IO from a single-source application p to a file X to another file Y. The timing of the routing and operation semantics is dictated by the control logic. For example, write requests could go to the new endpoint and reads could be controlled to go to the old or new endpoints. Endpoint routing enables functionality such as improving *tail latency* [Dean and Barroso 2013; Narayanan et al. 2008b], *copy-on-write* [Oracle 2010; Hitz et al. 1994; Rodeh et al. 2013], *file versioning* [Microsoft 2010], and *data re-encoding* [Abd-El-Malek et al. 2005]. These policies have in common the need for a dynamic mechanism that changes the endpoint of new data and routes IO

Table I. Examples of Specialized Functionality and the Type of IO Routing That Enables Them

	Functionality	How IO routing helps
Endpoint	Tail latency control	Route IO to less loaded servers
	Copy-on-write	Route writes to new location
	File versioning	Route IO to right version
Waypoint	Cache size guarantee	Route IO to specialized cache
	Deadline policies	Route IO to specialized scheduler
Scatter	Maximize throughput	Route `reads` to all replicas
	Minimize latency	Route `writes` to replica subset
	Logging/Debugging	Route selected IOs to loggers

to the appropriate endpoint. Section 5.1 shows how we implement tail latency control using endpoint routing.

Waypoint routing. Waypoint routing routes IO from a multi-source application $\{p, r\}$ to a file X through an intermediate waypoint stage W. W could be a file cache or scheduler. Waypoint routing enables specialized appliance processing [Sherry et al. 2012]. These policies need a dynamic mechanism to inject specialized waypoint stages or appliances along the stack and to selectively route IO to those stages. Section 5.3 shows how we implement file cache control using waypoint routing.

Scatter routing. Scatters IO from file X to additional endpoints Y and Z. The control logic dictates which subset of endpoints to read data from and write data to. Scatter routing enables specialized *replication* and *erasure coding* policies [Terry et al. 2013; Li et al. 2012], as well as interactive logging and debugging of the storage stack [Barham et al. 2004; Sigelman et al. 2010; Fonseca et al. 2007]. These policies have in common the need for a dynamic mechanism to choose which endpoint to write to and read from. This control enables programmable tradeoffs around throughput and update propagation latency. Section 5.2 shows how we implement replica set control using scatter routing. Section 5.4 shows how we implement performance debugging.

2.1. Challenges

IO routing is challenging for several reasons:

Consistent systemwide configuration updates. IO routing requires a control-plane mechanism for changing the path of an IO request. The mechanism needs to coordinate the forwarding rules in each sSwitch in the data plane. Any configuration changes must not lead to system instability, where an IO's semantic guarantees are violated by having it flow through an incorrect path.

Metadata consistency. IO routing allows `read` and `write` IOs to be sent to potentially different endpoints. Several applications benefit from this flexibility. Some applications, however, have stricter consistency requirements and require, for example, that a `read` always follow the path of a previous `write`. A challenge is keeping track of the data's latest location. Furthermore, IO routing metadata needs to coexist consistently with metadata in the rest of the system. The guest file system, for example, has a mapping of files to blocks, and the hypervisor has a mapping of blocks to virtual disks on an (often) remote storage backend. The backend could be a distributed system of its own with a metadata service mapping files or chunks to file systems to physical drives.

File system semantics. Some file system functionality (such as byte-range file locking when multiple clients access the same file) depends on consulting file system state to determine the success and semantics of individual IO operations. The logic and state that dictates the semantics of these operations resides inside the file system,

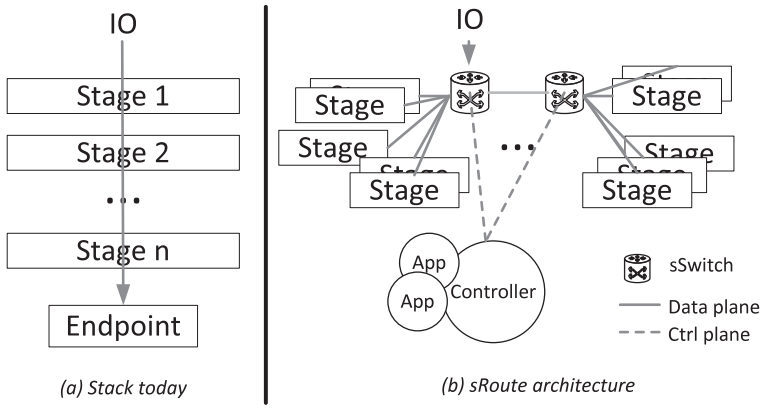

Fig. 2. System architecture. sSwitches can route IO within a physical machine's IO stack and across machines over the network.

Table II. Control API to the sSwitch

`Insert` (IOHeader, Delegate) Creates a new fwd. rule matching the IO header, using dynamic control delegate to look up destination
`Delete` (IOHeader) Deletes all rules matching the header
`Quiesce` (IOHeader, Boolean) Blocks or unblocks incoming IO matching IO header when Boolean is true or false respectively
`Drain` (IOHeader) Drains all pending IOs matching the IO header

at the destination endpoint of these IOs. IO routing needs to maintain the same file system functionality and semantics in the storage stack.

Efficiency. Providing IO stack customization requires a different way of building specialized functionality. We move away from an architecture that hard-codes functionality on the IO stack to an architecture that dynamically directs IOs to specialized stages. Any performance overheads incurred must be minimal.

3. DESIGN

Figure 2 shows sRoute's architecture. It is composed of the following:

—**sSwitches** on the data plane, that change the route of IOs according to forwarding rules. sSwitches are programmable through a simple API with four calls shown in Table II. The sSwitches forward IOs to other file destinations, the controller, or to specialized stages (e.g., one that implements a particular caching algorithm).
—**A control plane** with a logically centralized controller specifies the location of the sSwitches and inserts forwarding rules in them.
—**Specialized stages** that take an IO as an input, perform operations on its payload, and return the IO back to the sSwitch for further forwarding.

3.1. Baseline Architecture

The baseline system architecture our design builds on is that of an enterprise data center. Each tenant is allocated VMs or containers[1] and runs arbitrary applications or

[1]This article's implementation uses VMs.

Rule	:=	$IOHeader \rightarrow Delegate(IOHeader)$
IOHeader	:=	$< Source, Operation, File >$
Delegate	:=	$F(IOHeader); return\{Detour\}$
Source	:=	Unique Security Identifier
Operation	:=	`read`|`write`|`create`|`delete`
File	:=	$< FileName, Offset, Length >$
Detour	:=	$< IO \| IOHeader \| F(IOHeader), DetourLoc >$
DetourLoc	:=	$File \| Stage \| Controller$
Stage	:=	$< HostName, DriverName >$
F	:=	Restricted code

Fig. 3. Construct definitions.

services in them. Network and storage are virtualized and VMs are unaware of their topology and properties.

The baseline system is assumed to already have separate control and data planes and builds on the IOFlow architecture [Thereska et al. 2013]. That architecture provides support for flow-based classification and queuing and communication of basic per-flow statistics to a controller.

3.2. Design Goals

sRoute's design targets several goals. First, we want a solution that does not involve application or VM changes. Applications have limited visibility of the data center's IO stack. This article takes the view that data center services are better positioned for IO stack customization. These are then exposed to applications through new types of service level agreements, for example, guaranteeing better throughput and latency. Second, data-plane performance overheads should be minimal. Third, the control plane should be flexible and allow for a diverse set of application policies.

The rest of this section focuses on the sSwitches and the control plane interfaces to them. Section 4 presents implementation details. Section 5 focuses on control applications. Figure 3 provides the construct definitions used in the rest of the article.

3.3. sSwitches on the Data Plane

An sSwitch is a special stage that is inserted into the IO stack (data plane) to provide IO routing. An sSwitch forwards IO according to rules specified by the control plane. A forwarding rule contains two parts: an IO header and an action or delegate function.[2] IO packets are matched against the IO header, and the associated delegate in the *first* successful rule match executes (hence, the order of installed rules matters). In the simplest form, this delegate returns a set of stages where the IO should next be directed. For example, routing all traffic from VM_1 for file X on server S_1 to file Y on server S_2 can be represented with this rule:

1: $< VM_1, *, \ //S_1/X > \rightarrow (; return\{< IO, \ //S_2/Y >\})$.

An sSwitch implements four control plane API calls as shown in Table II. The APIs allow the control plane to `Insert` a forwarding rule or `Delete` it. Rules can be changed dynamically by two entities on the control plane: the controller or a control `Delegate` function.

As defined in Figure 3, the IO header is a tuple containing the source of an IO, the operation, and the file affected. The source of an IO can be a process or a VM uniquely

[2]The reason the second part of the rule is a function (as opposed to simply a set of routing locations) is for control plane efficiency in some situations, as is explained further in this section.

authenticated through a security identifier. The destination is a file in a (possibly remote) share or directory. Building on IOFlow's classification mechanism [Thereska et al. 2013] allows an sSwitch to have visibility over all this information and other relevant IO header entries at any point in the IO stack (without IOFlow, certain header entries such as the source, could be lost or overwritten as IO flows through the system).

The operation can be one of `read`, `write`, `create`, or `delete`. Wildcards and longest prefix matching can be used to find a match on the IO header. A default match rule sends an IO to its original destination. A detour location could be a file (e.g., another file on a different server from the original IO's destination), a stage on the path to the endpoint (example rule 1 below), or the centralized controller (example rule 2 below that sends the IO header for all writes from VM_2 to the controller):

1: $<VM_1, *, \ //S_1/X> \rightarrow (;return\{<IO, \ //S_2/C>\})$
2: $<VM_2, w, *> \rightarrow (;return\{<IOHeader, \ Controller>\})$.

The sSwitch is responsible for transmitting the full IO, its header, or the output of a function on the IO header to a set of stages. The response does not have to flow through the same path as the request, as long as it reaches the initiating source.[3]

Unlike in networking, the sSwitch needs to perform more work than just forwarding. It also needs to prepare the endpoint stages to accept IO, which is unique to storage. When a rule is first installed, the sSwitch needs to open a file connection to the endpoint stages, in anticipation of IO arriving. The sSwitch needs to create it and take care of any namespace conflicts with existing files. `Open` and `create` operations are expensive synchronous metadata operations. There is an inherent tradeoff between lazy file creation on the first IO arriving and file creation on rule installation. The former avoids unnecessarily creating files for rules that do not have any IO matching them, but on a match the first IO incurs a large latency. The latter avoids the latency but could create several empty files. The exact tradeoff penalties depend on the file systems used. By default, this article implements the latter, but ideally this decision would also be programmable (but it is not so yet.)

sSwitches implement two additional control plane APIs. A `Quiesce` call is used to block any further requests with the same IO header from propagating further. The implementation of this call builds on the lower-level IOFlow API that sets the token rate on a queue [Thereska et al. 2013]. `Drain` is called on open file handles to drain any pending IO requests downstream. Both calls are synchronous. These calls are needed to change the path of IOs in a consistent manner, as discussed in the next section.

3.4. Controller and Control Plane

A logically centralized controller has global visibility over the stage topology of the data center. This topology comprises all physical servers, network and storage components, as well as the software stages within a server. Maintaining this topology in a fault-tolerant manner is already feasible today [Isard 2007].

The controller is responsible for three tasks. First, it takes a high-level tenant policy and translates it into sSwitch API calls. Second, it decides *where* to insert the sSwitches and specialized stages in the IO stack to implement the policy. Third, it disseminates the forwarding rules to the sSwitches. We show these tasks step by step for two simple control applications below.

The first control application directs a tenant's IO to a *specialized file cache*. This policy is part of a case study detailed in Section 5.3. The tenant is distributed over

[3]sSwitches cannot direct IO responses to sources that did not initiate the IO. Finding scenarios that need such source routing and the mechanism for doing so is future work.

10 VMs on 10 different hypervisors and accesses a read-only dataset X. The controller forwards IO from this set of VMs to a specialized cache C residing on a remote machine connected to the hypervisors through a fast RDMA network. The controller knows the topology of the data paths from each VM to C and injects sSwitches at each hypervisor. It then programs each sSwitch as follows:

```
1: for i ← 1, 10 do
2:    Quiesce (<VM_i, *, //S_1/X>, true)
3:    Drain (<VM_i, *, //S_1/X>)
4:    Insert (<VM_i, *, //S_1/X>, (; return {<IO, //S_2/C>}))
5:    Quiesce (<VM_i, *, //S_1/X>, false).
```

Lines 2 and 3 are needed to complete any IOs in flight. This is done so the sSwitch does not need to keep any extra metadata to know which IOs are on the old path. That metadata would be needed, for example, to route a newly arriving `read` request to the old path since a previous `write` request might have been buffered in an old cache on that path. The delegate on line 4 simply returns the cache stage. Finally, line 5 unblocks IO traffic. The controller also injects an sSwitch at server S_2 where the specialized cache resides, so any requests that miss in cache are sent further to the file system of server S_1. The rule at S_2 matches IOs from C for file X and forwards them to server S_1:

```
1: Insert (<C, *, //S_1/X>, (; return {<IO, //S_1/X>}))
```

The second control application improves a tenant's **tail latency** and illustrates a more complex control delegate. The policy states that queue sizes across servers should be balanced. This policy is part of a case study detailed in Section 5.1. When a load burst arrives at a server S_1 from a source VM_1, the control application decides to temporarily forward that load to a less busy server S_2. The controller can choose to insert an sSwitch in the VM_1's hypervisor or at the storage server S_1. The latter means that IOs go to S_1 as before and S_1 forwards them to S_2. To avoid this extra network hop, the controller chooses the former. It then calls the following functions to insert rules in the sSwitch:

```
1: Insert (<VM_1, w, //S_1/X>, (F(); return {<IO, //S_2/X>}))
2: Insert (<VM_1, r, //S_1/X>, (; return {<IO, //S_1/X>})).
```

The rules specify that `writes` "w" are forwarded to the new server, whereas `reads` "r" are still forwarded to the old server. This application demands that `reads` return the latest version of the data. When subsequently a `write` for the first 512KB of data arrives,[4] the delegate function updates the read rule through function $F()$ whose body is shown below:

```
1: Delete (<VM_1, r, //S_1/X>)
2: Insert (<VM_1, r, //S_1/X, 0, 512KB >, (; return {<IO, //S_2/X>}))
3: Insert (<VM_1, r, //S_1/X>, (; return {<IO, //S_1/X>}).
```

Note that quiescing and draining are not needed in this scenario since the sSwitch is keeping the metadata necessary (in the form of new rules) to route a request correctly. A subsequent `read` for a range between 0 and 512KB will match the rule in line 2 and will be sent to S_2. Note that sSwitch matches on byte ranges as well, so a `read` for a

[4]The request's start offset and data length are part of the IO header.

range between 0 and 1024KB will be now split into two reads. The sSwitch maintains enough buffer space to coalesce the responses.

3.4.1. Delegates. The above examples showed instances of control delegates. Control delegates are restricted control plane functions that are installed at sSwitches for control plane efficiency. In the second example above, the path of an IO depends on the workload. `Write` requests can potentially change the location of a subsequent `read`. One way to handle this would be for all requests to be sent by the sSwitch to the controller using the following alternate rules and delegate function:

1: `Insert` (<VM_1, w, //S_1/X>, (; return {<IO, Controller>}))
2: `Insert` (<VM_1, r, //S_1/X>, (; return {<IO, Controller>})).

The controller would then serialize and forward them to the appropriate destination. Clearly, this is inefficient, bottlenecking the IO stack at the controller. Instead, the controller uses restricted delegate functions that make control decisions locally at the sSwitches.

This article assumes a non-malicious controller; however, the design imposes certain functionality restrictions on the delegates to help guard against accidental errors. In particular, delegate functions may only call the APIs in Table II and may otherwise only create or keep state local to the sSwitch. They may not rewrite the IO header or IO data. That is important since the IO header contains entries such as the source security descriptor that are needed for file access control to work in the rest of the system. These restrictions allow us to consider the delegates as a natural extension of the centralized controller. Simple programming language checks and passing the IO as read-only to the delegate enforce these restrictions.

3.5. Consistent Rule Updates

Forwarding rule updates could lead to instability in the system. This section introduces the notion of consistent rule updates. These updates preserve well-defined storage-specific properties. Similarly to networking [Reitblatt et al. 2012], storage has two different consistency requirements: per-IO and per-flow.

Per-IO consistency. Per-IO consistent updates require that each IO flows either through an old set of rules or an updated set of rules but not through a stack that is composed of old and new paths. The `Quiesce` and `Drain` calls in the API in Table II are sufficient to provide per-IO consistent updates.

Per-flow consistency. Many applications require a stream of IOs to behave consistently. For example, an application might require that a `read` request obtains the data from the latest previous `write` request. In cases where the same source sends both requests, then per-IO consistency also provides per-flow consistency. However, the second request can arrive from a different source, like a second VM in the distributed system. In several basic scenarios, it is sufficient for the centralized controller to serialize forwarding rule updates. The controller disseminates the rules to all sSwitches in two phases. In the first phase, the controller quiesces and drains requests going to the old paths and, in the second phase, the controller updates the forwarding rules.

However, a key challenge are scenarios where delegate functions create new rules. This complicates update consistency since serializing these new rules through the controller is inefficient when rules are created frequently (e.g., for every `write` request). In these cases, control applications attempt to provide all serialization through the sSwitches themselves. They do so as follows. First, they consult the topology map to identify points of serialization along the IO path. The topology map identifies common stages among multiple IO sources on their IO stack. For example, if two clients are

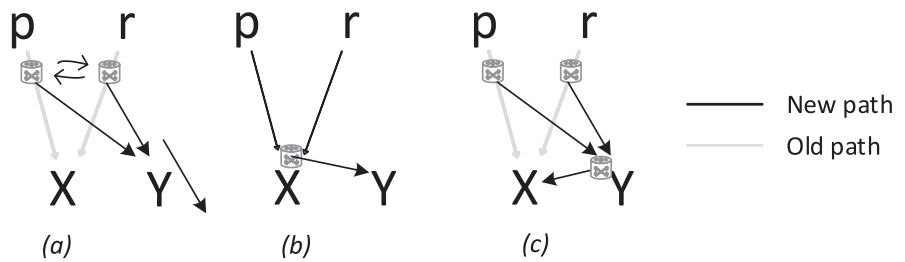

Fig. 4. Three possible options for placing sSwitches for consistent rule updates. Either can be chosen programmatically at runtime.

reading and writing to the same file X, then the control application has the option of inserting two sSwitches with delegate functions close to the two sources to direct both clients' IOs to Y. This option is shown in Figure 4(a). The sSwitches would then need to use two-phase commit between themselves to keep rules in sync, as shown in the figure. This localizes updates to participating sSwitches, thus avoiding the need for the controller to get involved.

A second option would be to insert a single sSwitch close to X (e.g., at the storage server) that forwards IO to Y. This option is shown in Figure 4(b). A third option would be to insert an sSwitch at Y that forwards IO back to X if the latest data are not on Y. This type of forwarding rule can be thought of as implementing *backpointers*. Note that two additional sSwitches are needed close to the source to forward all traffic, that is, reads and writes, to Y; however, these sSwitches do not need to perform two-phase commit. The choice between the last two options depends on the workload. If the control application expects that most IO will go to the new file, then the third option would eliminate an extra network hop.

3.6. Fault Tolerance and Availability

This section analyzes new potential risks on fault tolerance and availability induced by our system. Data continue to be N-way replicated for fault tolerance, and its fault tolerance is the same as in the original system.

First, the controller service is new in our architecture. The service can be replicated for availability using standard Paxos-like techniques [Lamport 1998]. If the controller is temporarily unavailable, then the implication on the rest of the system is at worst slower performance, but correctness is not affected. For example, IO that matches rules that require transmission to the controller will be blocked until the controller recovers.

Second, our design introduces new metadata in the form of forwarding rules at sSwitches. It is a design goal to maintain all state at sSwitches as soft state to simplify recovery—also there are cases where sSwitches do not have any local storage available to persist data. The controller itself persists all the forwarding rules before installing them at sSwitches. The controller can choose to replicate the forwarding rules, for example, using three-way replication (using storage space available to the controller—either locally or remotely).

However, forwarding rules created at the control delegates pose a challenge because they need persisting. sRoute has two options to address this challenge. The first is for the controller to receive all delegate updates synchronously, ensure they are persisted, and then return control to the delegate function. This option involves the controller on the critical path. The second option (the default) is for the delegate rules to be stored with the forwarded IO data. A small header is prepended to each IO containing the updated rule. On sSwitch failure, the controller knows to which servers IO has been forwarded and recovers all persisted forwarding rules from them.

Third, sSwitches introduce new code along the IO stack, thus increasing its complexity. When sSwitches are implemented in the kernel (see Section 4), an sSwitch failure may cause the entire server to fail. We have kept the code footprint of sSwitches small and we plan to investigate software verification techniques in the future to guard against such failures.

3.7. Design Considerations

The behaviour and semantics of some file system operations (e.g., file locking) is determined by state stored inside the file system. Because IO routing is done inside the storage stack, above the file system layer, maintaining the semantics of these operations introduces extra complexity to our design. Currently, there are two solutions to maintaining the semantics of these operations: (i) the state can remain in the file system at the original endpoint, and sSwitches can issue RPCs to query it as IOs flow through the system (naturally, this is not desirable due to the extra communication overhead per IO) or (ii) when an sSwitch begins routing a flow, the relevant file system state can be pushed into the sSwitch, and subsequently maintained there, such that the sSwitch can maintain the desired semantics for subsequent incoming IOs.

4. IMPLEMENTATION

An sSwitch is implemented partly in kernel level and partly in user level. The kernel part is written in C and its functionality is limited to partial IO classification through longest prefix matching and forwarding within the same server. The user-level part is written in C# and implements further sub-file-range classification using hash tables. It also implements forwarding IO to remote servers. An sSwitch is a total of 25 kLOC. The kernel part of our implementation has been released and is free to download as part of the MSR Storage Toolkit [Research 2014].

Routing within a server's IO stack. Our implementation makes use of the filter driver architecture in Windows [Microsoft Corporation 2014b]. Each filter driver implements a stage in the kernel and is uniquely identified using an altitude ID in the IO stack. The kernel part of the sSwitch automatically attaches control code to the beginning of each filter driver processing. Bypassing a stage is done by simply returning from the driver early. Going through a stage means going through all the driver code.

Routing across remote servers. To route an IO to an arbitrary remote server's stage, the kernel part of the sSwitch first performs an upcall sending the IO to the user-level part of the sSwitch. That part then transmits the IO to a remote detour location using TCP or RDMA (default) through the SMB file system protocol. On the remote server, an sSwitch intercepts the arriving packet and routes it to a stage within that server.

sSwitch and stage identifiers. An sSwitch is a stage and has the same type of identifier. A stage is identified by a server host name and a driver name. The driver name is a tuple of <device driver name, device name, altitude>. The altitude is an index into the set of drivers or user-level stages attached to a device.

Other implementation details. For the case studies in this article, it has been sufficient to inject one sSwitch inside the Hyper-V hypervisor in Windows and another on the IO stack of a remote storage server just above the NTFS file system using file system filter drivers [Microsoft Corporation 2014b]. Specialized functionality is implemented entirely in user-level stages in C#. For example, we have implemented a user-level cache (Section 5.3). The controller is also implemented in user level and communicates with both kernel- and user-level stages through RPCs over TCP. Routing happens on a per-file basis at block granularity. Our use cases do not employ any semantic information about the data stored in each block. For control applications that require such information, the functionality would be straightforward to implement,

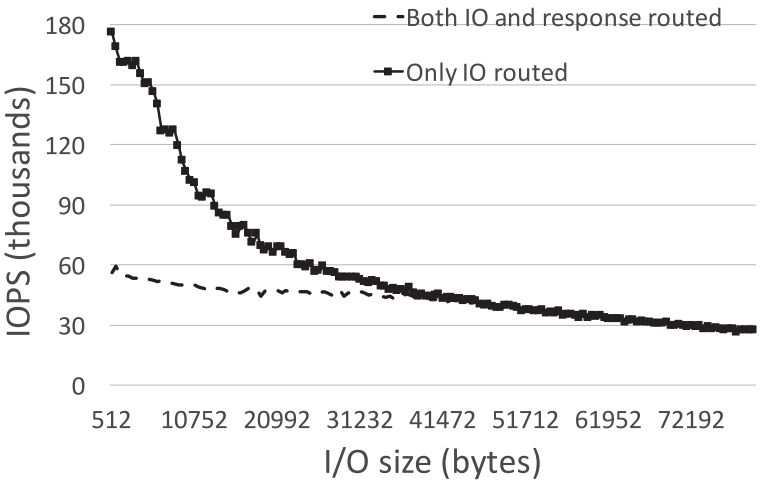

Fig. 5. Current performance range of an sSwitch.

using miniport [Microsoft Corporation 2014a] drivers, instead of filter drivers. Applications and VMs always run unmodified on our system. However, some applications pass several static hints such as "write through" to the OS using hard-coded flags. The sSwitches intercept `open/create` calls and can change these flags. In particular, for specialized caching (Section 5.3) the sSwitches disable OS caching by specifying `Write-through` and `No-buffering` flags. Caching is then implemented through the control application. To avoid namespace conflict with existing files, sRoute stores files in a reserved "sroute-folder" directory on each server. That directory is exposed to the cluster as an SMB share writable by internal processes only.

Implementation limitations. A current limitation of the implementation is that sSwitches cannot intercept individual IO to memory mapped files. However, they can intercept bulk IO that loads a file to memory and writes pages to disk, which is sufficient for most scenarios.

Another current limitation of our implementation is that it does not support byte-range file locking for multiple clients accessing the same file, while performing endpoint routing. The state to support this functionality is kept in the file system, at the original endpoint of the flow. When the endpoint is changed, this state is unavailable. To support this functionality, there are two alternatives, as described in Section 3.7.

The performance range of the current implementation of an sSwitch is illustrated in Figure 5. This throughput includes passing an IO through both kernel and user-level. Two scenarios are shown. In the "Only IO routed" scenario, each IO has a routing rule, but an IO's response is not intercepted by the sSwitch (the response goes straight to the source). In the "Both IO and response routed" scenario both an IO and its response are intercepted by the sSwitch. Intercepting responses is important when the response needs to be routed to a non-default source as well (one of our case studies for caches in Section 5.3 requires response routing). Intercepting an IO's response in Windows is costly (due to interrupt handling logic beyond the scope of this article), and the performance difference is a result of the OS, not of the sSwitch. Thus, the performance range for small IO is between 50,000 and 180,000 IOPS, which makes sSwitches appropriate for an IO stack that uses disk or SSD backends but not yet a memory-based stack.

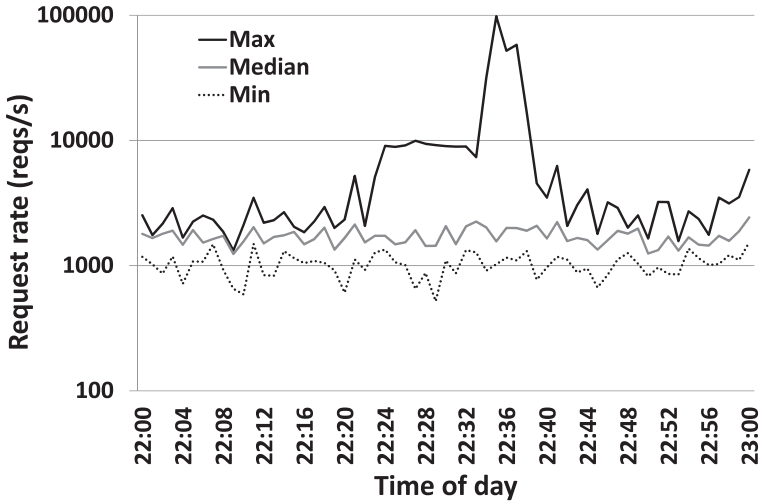

Fig. 6. Load on three Exchange server volumes showing load imbalances.

5. CONTROL APPLICATIONS

This section makes three points. First, we show that a diverse set of control applications can be built on top of IO routing. Thus, we show that the programmable routing abstraction can replace one-off hard-coded implementations. We have built and evaluated four control applications implementing tail latency control, replica set control, file cache control, and performance debugging. These applications cover each of the detouring types in Table I. Second, we show that tenants benefit significantly from the IO customization provided by the control applications. Third, we evaluate data and control plane performance.

Testbed. The experiments are run on a testbed with 12 servers, each with 16 Intel Xeon 2.4GHz cores, 384GB of RAM, and Seagate Constellation 2 disks. The servers run the Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40Gbps Mellanox ConnectX-3 NIC supporting RDMA and connected to a Mellanox MSX1036B-1SFR switch.

Workloads. We use three different workloads in this section. The first is TPC-E [Transaction Processing Performance Council 2014] running over unmodified SQL Server 2012 R2 databases. TPC-E is a transaction processing OLTP workload with small IO sizes. The second workload is a public IO trace from an enterprise Exchange email server [SNIA 2007]. The third workload is IoMeter [Intel Corporation 2014], which we use for controlled micro-benchmarks.

5.1. Tail Latency Control

Tail latency in data centers can be orders of magnitude higher than average latency leading to application unresponsiveness [Dean and Barroso 2013]. One of the reasons for high tail latency is that IOs often arrive in bursts. Figure 6 illustrates this behavior in publicly available Exchange server traces [SNIA 2007], showing traffic to three different volumes of the Exchange trace. The difference in load between the most loaded volume and the least loaded volume is two orders of magnitude and lasts for more than 15min.

Data center providers have load balancing solutions for CPU and network traffic [Greenberg et al. 2009]. IO to storage, on the other hand, is difficult to load balance at short timescales because it is stateful. An IO to an overloaded server S must go to

S since it changes state there. The first control application addresses the tail latency problem by temporarily forwarding IOs from loaded servers onto less loaded ones while ensuring that a `read` always accesses the last acknowledged update. This is a type of *endpoint routing*. The functionality provided is similar to Everest [Narayanan et al. 2008b] but written as a control application that decides when and where to forward to based on global system visibility.

The control application attempts to balance queue sizes at each of the storage servers. To do so, for each storage server, the controller maintains two running averages based on stats it receives[5]: Req_{Avg} and Req_{Rec}. Req_{Avg} is an exponential moving average over the last hour. Req_{Rec} is an average over a sliding window of 1min, meant to capture the workload's recent request rate. The controller then temporarily forwards IO if:

$$Req_{Rec} > \alpha Req_{Avg},$$

where α represents the relative increase in request rate that triggers the forwarding. We evaluate the impact of this control application on the Exchange server traces shown in Figure 6, but first we show how we map this scenario into forwarding rules.

There are three flows in this experiment. Three different VMs, VM_{max}, VM_{min}, and VM_{med}, on different hypervisors access one of the three volumes in the trace "Max," "Min," and "Median." Each volume is mapped to a Virtual Hard Drive (VHD) file VHD_{max}, VHD_{min}, and VHD_{med}, residing on three different servers S_{max}, S_{min}, and S_{med}, respectively. When the controller detects imbalanced load, it forwards write IOs from the VM accessing S_{max} to a temporary file T on server S_{min}:

1: $<*, w,\ //S_{max}/VHD_{max} >\rightarrow (F(); return\{<IO,\ //S_{min}/T>\})$
2: $<*, r,\ //S_{max}/VHD_{max} >\rightarrow (; return\{<IO,\ //S_{max}/VHD_{max}>\}).$

Read IOs follow the path to the most up-to-date data, whose location is updated by the delegate function $F()$ as the write IOs flow through the system. We showed how $F()$ updates the rules in Section 3.4. Thus, the forwarding rules always point a read to the latest version of the data. If no writes have happened yet, then all reads by definition go to the old server VM_{max}.

The control application may also place a specialized stage O in the new path that implements an optional log-structured layout that converts all writes to streaming writes by writing them sequentially to S_{min}. The layout is optional since SSDs already implement it internally and it is most useful for disk-based backends. The control application inserts a rule forwarding IO from the VM first to O (rule 1 below) and another to route from O to S_{min} (rule 2).

1: $<*, *,\ //S_{max}/VHD_{max} >\rightarrow (; return\{<IO,\ //S_{min}/O>\})$
2: $<O, *,\ //S_{max}/VHD_{max} >\rightarrow (; return\{<IO,\ //S_{min}/T>\}).$

Note that in this example data are partitioned across VMs and no VMs share data. Hence, the delegate function in the sSwitch is the only necessary point of metadata serialization in system. This is a simple version of case (a) in Figure 4 where sSwitches do not need two-phase commit. The delegate metadata are temporary. When the controller detects that a load spike has ended, it triggers data *reclaim*. All sSwitch rules for writes are changed to point to the original file VHD_{max}. Note that read rules still point to T until new arriving writes overwrite those rules to point to VHD_{max} through their delegate functions. The controller can optionally speed up the reclaim process by actively copying forwarded data to its original location. When the reclaim process ends, all rules can be deleted, and the the sSwitches and specialized stage removed from the

[5]The controller uses IOFlow's `getQueueStats` API [Thereska et al. 2013] to gather systemwide statistics for all control applications.

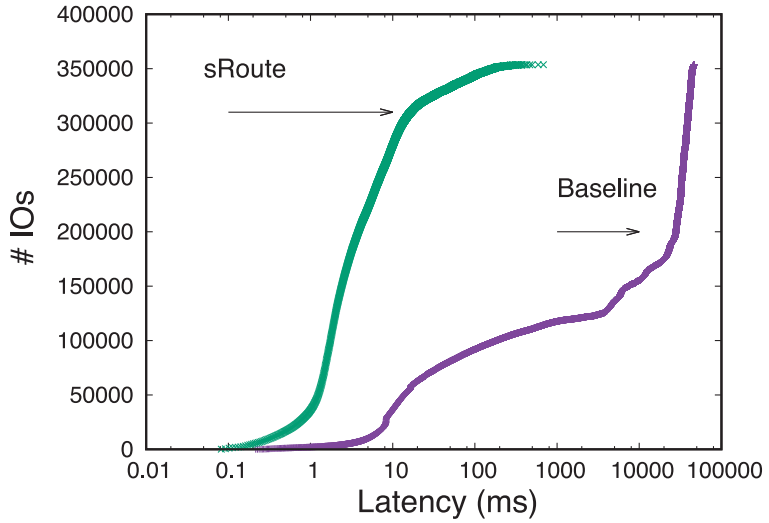

Fig. 7. CDF of response time for baseline system and with IO routing.

IO stack, since all data reside in and can be accessed again from the original server S_{max}.

We experiment by replaying the Exchange traces using a time-accurate trace replayer on the disk-based testbed. We replay a 30min segment of the trace, capturing the peak interval and allowing for all forwarded data to be reclaimed. We also employ a log-structured write optimization stage that converts all writes to sequential writes. Figure 7 shows the results. IO routing results in two orders of magnitude improvements in tail latency for the flow to S_{max}. The change latency distribution for S_{min} (not shown) is negligible.

Overheads. Data (2.8GB) were forwarded and the delegate functions persisted approximately 100,000 new control plane rules with no noticeable overhead. We experimentally triggered one sSwitch failure and measured that it took approximately 30s to recover the rules from the storage server. The performance benefit obtained is similar to specialized implementations [Narayanan et al. 2008b]. The CPU overhead at the controller was less than 1%.

5.2. Replica Set Control

No one replication protocol fits all workloads [Abd-El-Malek et al. 2005; Terry et al. 2013; Li et al. 2012]. Data center services tend to implement one particular choice (e.g., primary-based serialization) and offer it to all workloads passing through the stack (e.g., Calder et al. [2011]). One particularly important decision that such an implementation hard codes is the choice of write-set and read-set for a workload. The write-set specifies the number of servers to contact for a `write` request. The size of the write-set has implications on request latency (a larger set usually means larger latency). The read-set specifies the number of servers to contact for `read` requests. A larger read-set usually leads to higher throughput since multiple servers are read in parallel.

The write- and read-sets need to intersect in certain ways to guarantee a chosen level of consistency. For example, in primary-secondary replication, the intersection of the write- and read-sets contains just the primary server. The primary then writes the

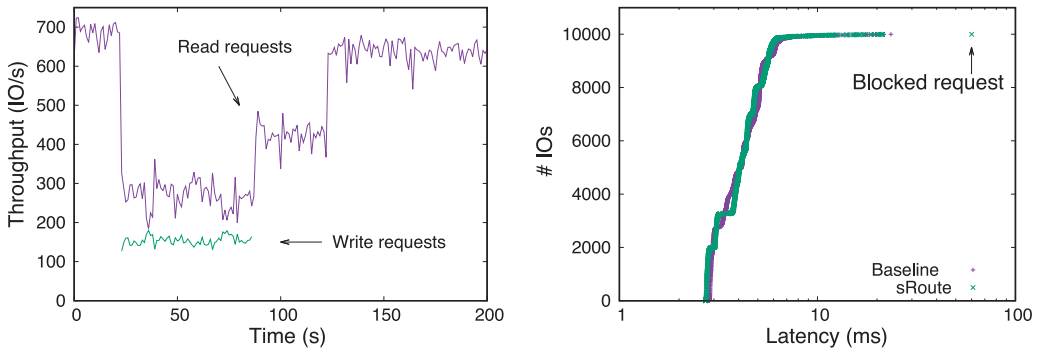

Fig. 8. Reads benefit from parallelism during read-only phases and the system performs correct serialization during read:write phases (left). The first write needs to block until forwarding rules are changed (right).

data to a write-set containing the secondaries. The request is completed once a subset of the write-set has acknowledged it (the entire write-set by default).

The replica set control application provides a configurable write- and read-set. It uses only *scatter routing* to do so, without any specialized stages. In the next experiment the policy at the control application specifies that if the workload is read-only, then the read-set should be all replicas. However, for correct serialization, if the workload contains writes, all requests must be serialized through the primary, that is, the read-set should be just the primary replica. In this experiment, the application consists of 10 IoMeters on 10 different hypervisors reading and writing to a 16GB file using two-way primary-based replication on the disk testbed. IoMeter uses 4KB random-access requests and each IoMeter maintains four requests outstanding.

The control application monitors the read:write ratio of the workload through the `getQueueStats` API call, and when it detects that it has been read-only for more than 30s (a configurable parameter), it switches the read-set to be all replicas. To do that, it injects sSwitches at each hypervisor and sets up rules to forward reads to a randomly chosen server S_{rand}. This is done through a control delegate $F()$ that picks the next server at random. To make the switch between old and new rule, the controller firsts `quiesces` writes and then `drains` them. It then inserts the new read-set rule (rule 1) as follows:

1: $<*, r,\ //S_1/X> \rightarrow (F(); return\{<IO,\ //S_{rand}/X>\})$
2: $<*, w, *> \rightarrow (; return\{<IOHeader,\ Controller>\}).$

The controller is notified of the arrival of any write requests by the rule (2). The controller then proceeds to revert the read-set rule and restarts the write stream.

Figure 8 shows the results. The performance starts high since the workload is in a read-only state. When the first write arrives at time 25, the controller switches the read-set to contain just the primary. In the third phase starting at time 90, writes complete and read performance improves, since reads do not contend with writes. In the fourth phase at time 125, the controller switches the read-set to be both replicas, improving read performance by 63% as seen in Figure 8 (left). The tradeoff is that the first write requests that arrive incur a latency overhead from being temporarily blocked while the write is signalled to the controller, as shown in Figure 8 (right). Depending on the application performance needs, this latency overhead can be amortized appropriately by increasing the time interval before assuming the workload is read-only. The best-case performance improvement expected is 2×, but the application (IoMeter) has a low number of outstanding requests and does not saturate storage in this example.

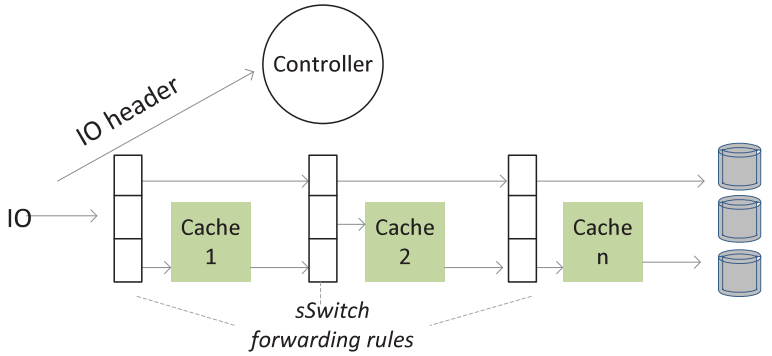

Fig. 9. Controller sets path of an IO through multiple cache using forwarding rules in sSwitches.

Overheads. The control application changes the forwarding rules infrequently at most every 30s. In an unoptimized implementation, a rule change translated to 418Bytes/flow for updates (40MB for 100,000 flows). The control application received stats every second using 302Bytes/flow for statistics (29MB/s for 100,000 flows). The CPU overhead at the controller is negligible.

5.3. File Cache Control

File caches are important for performance: Access to data in the cache is more than 3 orders of magnitude faster than to disks. A well-known problem is that data center tenants today have no control over the location of these caches or their policies [Arpaci-Dusseau and Arpaci-Dusseau 2001; Engler et al. 1995; Cao et al. 1996; Stefanovici et al. 2015]. The only abstraction the data center provides to a tenant today is a VM's memory size. This is inadequate in capturing all the places in the IO stack where memory could be allocated. VMs are inadequate even in providing isolation: An aggressive application within a VM can destroy the cache locality of another application within that VM.

Previous work [Stefanovici et al. 2015] has explored the programmability of caches on the IO stack and showed that applications and cloud providers can greatly benefit from the ability to customize cache size, eviction, and write policies, as well as explicitly control the placement of data in caches along the IO stack. Such explicit control can be achieved by using filter rules installed in a cache [Stefanovici et al. 2015]. All incoming IO headers are matched against installed filter rules, and an IO is cached if its header matches an installed rule. However, this type of simple control only allows IOs to be cached at some point along their *fixed* path from the application to the storage server. The ability to route IOs to arbitrary locations in the system using sSwitches while maintaining desired consistency semantics allows *disaggregation* of cache memory from the rest of a workload's allocated resources.

This next file cache control application provides several IO stack customizations through *waypoint routing*. We focus on one here: cache isolation among tenants. Cache isolation in this context means that (a) the controller determines how much cache each tenant needs, and (b) the sSwitches isolate one tenant's cache from another's. sRoute controls the path of an IO. It can forward an IO to a particular cache on the data plane. It can also forward an IO to bypass a cache as shown in Figure 9.

The experiment uses two workloads, TPC-E and IoMeter, competing for a storage server's cache. The storage backend consists of hard disks. The TPC-E workload represents queries from an SQL Server database with a footprint of 10GB running within a VM. IoMeter is a random-access read workload with IO sizes of 512KB. sRoute's policy in this example is to maximize the utilization of the cache with the hit rate measured

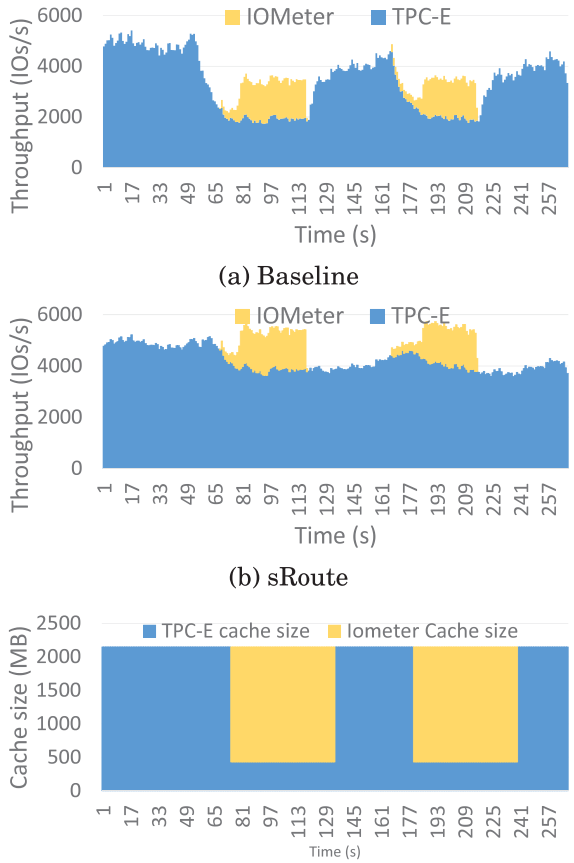

Fig. 10. Maximizing hit rate for two tenants with different cache miss curves.

in terms of IOPS. In the first step, all IO headers are sent to the controller, which computes their miss ratio curves using a technique similar to SHARDS [Waldspurger et al. 2015].

Then the controller sets up sSwitches so the IOs from IOMeter and from TPC-E go to different caches $C_{IOMeter}$ and C_{TPCE} with sizes provided by SHARDS, respectively (the caches reside at the storage server):

1: $<IOMeter, *, *>, (; return\{<IO, C_{IOMeter}>\})$
2: $<TPCE, *, *>, (; return\{<IO, C_{TPCE}>\})$.

Figure 10 shows the performance of TPC-E when competing with two bursts of activity from the IoMeter workload, with and without sRoute. When sRoute is enabled (Figure 10(b)), total throughput increases when both workloads run. In contrast, with today's caching (Figure 10(a)), total throughput actually drops. This is because IoMeter takes enough cache away from TPC-E to displace its working set out of the cache. With sRoute, total throughput improves by 57% when both workloads run, and TPC-E's performance improves by 2×.

Figure 10(c) shows the cache allocations output by our control algorithm when sRoute is enabled. Whenever IoMeter runs, the controller gives it 3/4 of the cache, whereas

TPC-E receives 1/4 of the cache, based on their predicted miss ratio curves. This cache allocation leads to each receiving around 40% cache hit ratio. Indeed, the allocation follows the miss ratio curve that denotes what the working set of the TPC-E workload is—after this point diminishing returns can be achieved by providing more cache to this workload. Notice that the controller apportions unused cache to the TPC-E workload 15s after the IoMeter workload goes idle.

Overheads. The control application inserted forwarding rules at the storage server. Rule changes were infrequent (the most frequent was every 30s). The control plane uses approximately 178Bytes/flow for rule updates (17MB for 100,000 flows). The control plane subsequently collects statistics from sSwitches and cache stages every control interval (default is 1s). The statistics are around 456Bytes/flow (roughly 43MB for 100,000 flows). We believe these are reasonable control plane overheads. Our current method for generating miss ratio curves (a non-optimized variant of SHARDS) runs offline and consumes 100% of two cores at the controller.

5.4. Performance Debugging

In a data center, IOs from applications traverse numerous software stages across several physical machines on their way to the durable storage backend. In such an environment, applications can experience degraded storage performance, due to short- and long-lived bottlenecks appearing at various points in the stack. This can occur for a variety of reasons, such as the following: the available storage capacity cannot match the changing demands of the application, shared resources (CPU, memory, network) can become heavily contended without proper resource isolation, or data reconstruction (such as RAID recovery) can introduce extra load on the storage backend after a hardware failure.

Unfortunately, despite the multitude of causes, degraded storage performance exhibits in the same way to applications: large increases in IO latencies and drops in storage throughput. In an environment where the storage stack spans several machines and consists of many software stages, it is difficult (if not impossible) for operators to dynamically pinpoint and address bottlenecks along the IO path quickly.

The control application described in this section enables performance debugging in the IO stack by leveraging the control delegate functions while performing any type of IO routing. The control delegates measure individual IO response times at multiple points in the stack and sends them to the controller, who then calculates the latencies incurred by IOs between each stage to pinpoint bottlenecks in the stack.

To measure individual IO response times, a control delegate at an sSwitch records the timestamp when the IO was first seen into a hash table. This state is maintained until the IO is seen again on its return path, at which point its response time is calculated and sent to the controller, as shown in Figure 11. To see how an sSwitch accomplishes this, we consider the one at stage A, which routes IOs for the file X according to the following rule:

```
1: <*, *, //S1/X> → (F(IOHeader);
     return{<IO, B>, <F(IOHeader), Controller>}).
```

The rule states that a delegate function $F()$ is first executed, and then the IO is forwarded to the next stage B, while the output of the delegate function (the IO's response time) is sent to the controller. Since the delegate function $F()$ needs to intercept an IO on both its outgoing and return path, the delegate function's body is split into two parts. The first executes when the IO is first seen on its outgoing path:

```
1: timestamps[IOHeader] = Time.Now;
```

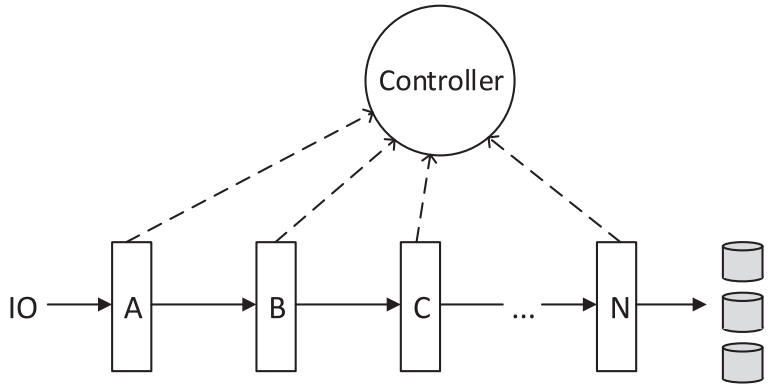

Fig. 11. sSwitches send their measured IO response times for the IO going to file X to the controller.

The second part executes on the IO's return path:

1: $responseTimes[IOHeader] = Time.Now - timestamps[IOHeader]$;
2: $return\ responseTimes[IOHeader]$;

sSwitches along the IO path each collect the observed response time for an IO from their position in the stack, and send them to the controller, generating a sequence:

$$resp_A, resp_B, \ldots, resp_N.$$

The controller then pairwise subtracts the reponse times to derive the latency of each link on the IO path as follows:

$$\begin{aligned} latency_{AB} &= resp_A - resp_B \\ latency_{BC} &= resp_B - resp_C \\ &\ldots \\ latency_{(N-1)N} &= resp_{(N-1)} - resp_N \end{aligned}.$$

Having calculated the latencies for each of the links along the IO path, the controller can then easily identify the bottlenecked link that is causing the application to experience increases in IO latencies. The data center operator can then focus their attention on that link to mitigate the existing problem. This control application provides functionality for storage similar to that provided by the *traceroute* utility for networks.

Since collecting response times for in-flight IOs might introduce extra latency on the data path, the control application can also sample just a fraction of IOs, reducing the impact of debugging on the live system. This sampling can be turned on or off, as well as dynamically adjusted at runtime.

Overheads. Figure 12 quantifies the overhead of collecting response times for in-flight IOs in an sSwitch. In order to quantify worst-case performance, we used IOMeter to perform sequential Read operations from an SSD backend, varying the IO size between 0.5KB and 64KB and varying the sample rate from 0% (no response times collected) to 100% (response times collected for every IO). Note that across all IO sizes, even at 100% sampling, the overheads are never move than 20% for latency and 17% for throughput. When taking confidence intervals into account, the differences are not significant. We believe these overheads are acceptable, as they are only incurred while temporarily diagnosing problems that otherwise cause orders-of-magnitude drops in performance. In addition, sampling response times can be turned off after enough samples have been collected and the problem is being diagnosed. Currently, there is considerably more variance in the overhead costs for higher IO sizes, and we are working to minimize this, as well as the overall performance overhead.

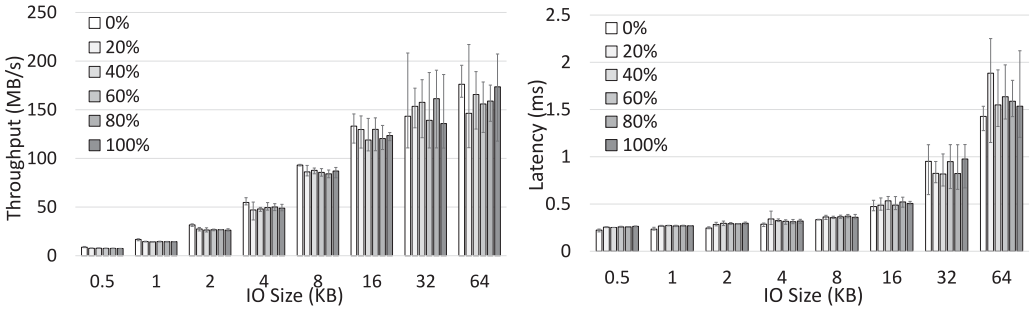

Fig. 12. Average overheads of collecting response times while performance debugging. Each bar represents a different sampling rate, with 0% baseline, and 100% when sampling every IO. Error bars show minimum and maximum values over three runs.

6. OPEN QUESTIONS

In this section, we outline open questions and directions for future work on software-defined storage. We hope this will generate interest and encourage future research in the area.

—sRoute currently lacks any verification tools that could help programmers. For example, it is possible to write incorrect control applications that route IOs to arbitrary locations, resulting in data loss. Thus, the routing flexibility is powerful but unchecked. There are well-known approaches in networking, such as header space analysis [Kazemian et al. 2013], that we believe could also apply to storage, but we have not investigated them yet. Similar approaches could also be used to make sure that routing rules from multiple control applications running on the same controller co-exist in the system safely and that they collectively achieve the correct desired functionality.

—SDN has become a well-established field over the past several years. As a community, we have gained experience with SDN controllers and following this work also with software-defined storage (SDS) controllers. It would be desirable to have a control plane that understands both the network and storage. For example, it is currently possible to get inconsistent end-to-end policies, where the storage controller decides to send data from server A to B, while the network controller decides to block any data from A going to B. Unifying the control plane across resources is an important area for future work.

—It would also be interesting to explore domain-specific languages and to specify and control storage functionality built using the lower-level programmable storage primitives explored in this article. The goal would be to raise the level of abstraction for programmers using our system and make it easier to specify storage policies and functionality at a higher level rather than directly controlling the lower-level mechanisms used to implement this functionality.

—IO routing is currently done in the IO stack, without application involvement. As such, we maintain the application-facing interface and semantics of the current IO stack, while opening up the stack and making it programmable by data center operators. While this enables a wealth of new functionality and policies to be implemented (some of which are described in this article), there may be other functionality and policies that could benefit from some amount of application involvement (or greater knowledge of application semantics). For example, performing a backup requires obtaining a consistent snapshot of the application. This is supported in our current design, where the backup is initiated by the client application. However, any policy that involves scheduling or initiating backups based on global state across several

machines (e.g., current network load or storage server idleness) will require some amount of knowledge about application state and potentially direct cooperation from the application. Exploring application involvement in IO routing and the types of policies and functionality it enables is an interesting area for future work. Similar recent work has explored application involvement for data center networks [Ballani et al. 2015].

—Another interesting area of exploration relates to handling policies for storage data at rest. Currently, sRoute operates on IO as it is flowing through the system. Once the IO reaches its destination, it is considered at rest. It might be advantageous for an sSwitch itself to initiate data movement for data at rest. That would require new forwarding rule types and make an sSwitch more powerful.

—Previous work [Angel et al. 2014] has shown how to dynamically infer application demands, as well as data center appliance capacities, and enforce end-to-end performance isolation across an entire data path comprising several machines. It is important to incorporate knowledge of data path capacity into the control logic that makes routing decisions. Understanding the feasibility and impact of changing the path of a flow of IOs can have a significant impact on overall performance and on quickly adapting to changes in demand or capacity.

—There recently has been an increasing focus on streaming IO, where data do not rest (in a file or database) but are constantly flowing at high rates from one processing node to another. Features are extracted from such data for analytics and machine learning, and decisions need to be made on the fly as to where a record goes next. Streaming IO is currenty handled by a different storage stack to traditional file or record-based data, leading to inefficiencies from having two separate ways to store and analyse data. It is an open research question whether such inefficiencies could be resolved by one storage stack. One particularly interesting problem has to do with the granularity of the classification and routing mechanisms. Previous work [Thereska et al. 2013] on SDS has focused on traditional file-based storage workloads. For those workloads, a file (e.g., a VHD image) stays open for the duration of the workload, and the IO classification and routing stays static for the duration of the workload, which can be minutes, hours, or even days. Streaming IO changes some of these fundamental workload characteristics. IO classification and routing often needs to be done per-IO rather than per-file-open session. The IO sizes tend to be smaller than traditional file-based workloads (i.e., <512Bytes), which means that per-IO handling needs to be efficient. IOs do not necessarily terminate in files. Often they terminate in a sink processor node that transmits them to another device for further processing (typically over TCP sockets). Hence, the challenge is to extend the SDS language for specifying sources and sinks to accomodate a wider range of possibilities. At a high level, because streaming IO has similar characteristics to networked IO, this further hints at an opportunity to unify SDN and SDS into one hyper-converged framework.

—Another emerging area of research is on the use of programable FPGA hardware in data centers to speed up various aspects of cloud workloads such as web search, machine learning, and network processing [Putnam et al. 2014; Greenberg 2015; Netronome 2016]. It would be interesting to explore the offload of storage functionality currently implemented in inlined software stages onto FPGAs. This is a particularly intriguing as data center FPGA use is already strongly coupled with the network [Greenberg 2015; Netronome 2016], further lending to the opportunity of a hyper-converged SDN/SDS network.

7. RELATED WORK

Our work is most related to SDNs [Koponen et al. 2010; Casado et al. 2007; Yan et al. 2007; Ferguson et al. 2013; Qazi et al. 2013; Jain et al. 2013; Tolia et al. 2006;

Cully et al. 2014] and SDS [Arpaci-Dusseau and Arpaci-Dusseau 2001; Thereska et al. 2013]. Specifically, our work builds directly on the control-data decoupling enabled by IOFlow [Thereska et al. 2013] and borrows two specific primitives: classification and rate limiting based on IO headers for quiescing. IOFlow also made a case for request routing. However, it only explored the concept for bypassing stages along the path and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This chapter provides the full routing abstraction.

There has been much work in providing applications with specialized use of system resources [Engler et al. 1995; Kaashoek et al. 1997; Bershad et al. 1995; Arpaci-Dusseau and Arpaci-Dusseau 2001; Arpaci-Dusseau et al. 2003]. The Exokernel architecture [Engler et al. 1995; Kaashoek et al. 1997] provides applications with direct control over resources with minimal kernel involvement. SPIN [Bershad et al. 1995] and Vino [Seltzer et al. 1996] allow applications to download code into the kernel and specialize resource management for their needs. Another orthogonal approach is to extend existing OS interfaces and pass hints vertically along the IO stack [Arpaci-Dusseau and Arpaci-Dusseau 2001; Arpaci-Dusseau et al. 2003, 2006; Mesnier et al. 2011]. Hints can be passed in both directions between the application and the system, exposing application needs and system resource capabilities to provide a measure of specialization.

In contrast to the above approaches, this chapter makes the observation that modern IO stacks support mechanisms for injecting stages with specialized functionality (e.g., in Windows [Microsoft Corporation 2014b], FreeBSD [FreeBSD 2014], and Linux [Love 2010]). sRoute transforms the problem of providing application flexibility into an IO routing problem. sRoute provides a control plane to customize an IO stack by forwarding IO to the right stages without changing the application or requiring a different OS structure.

We built three control applications on top of IO routing. The functionality provided from each has been extensively studied in isolation. For example, application-specific file cache management has shown significant performance benefits [Cao et al. 1996; Harty and Cheriton 1992; Krueger et al. 1993; Wong and Wilkes 2002; Huang et al. 2013; Stefanovici et al. 2015]. Snapshots, copy-on-write, and file versioning all have at their core IO routing. Hard-coded implementations can be found in file systems like ZFS [Oracle 2010], WAFL [Hitz et al. 1994], and btrfs [Rodeh et al. 2013]. Similarly, Narayanan et al. describe an implementation of load balancing through IO offloading of write requests [Narayanan et al. 2008a, 2008b]. Abd-el-malek et al. describe a system implementation where data can be re-encoded and placed on different servers [Abd-El-Malek et al. 2005]. Finally, several distributed storage systems each offer different consistency guarantees [Baker et al. 2011; Cooper et al. 2008; Terry et al. 2013; DeCandia et al. 2007; Li et al. 2012; Corbett et al. 2012; Lakshman and Malik 2010; Terry et al. 1995; Calder et al. 2011; Chang et al. 2006]. In contrast to these specialized implementations, sRoute offers a programmable IO routing abstraction that allows for all this functionality to be specified and customized at runtime.

8. CONCLUSION

This article presents sRoute, an architecture that enables an IO routing abstraction and makes the case that it is useful. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. This article shows how sRoute can provide unmodified applications with specialized tail latency control, replica set control, achieve file cache isolation, and aid in performance

debugging, all to substantial benefit. We also outlined several promising avenues for future work, which we hope will motivate others to contribute to this area.

ACKNOWLEDGMENTS

We thank the anonymous FAST reviewers; our shepherd, Jason Flinn; and others, including Hitesh Ballani, Thomas Karagiannis, and Antony Rowstron, for their feedback.

REFERENCES

Michael Abd-El-Malek, William V. Courtright, II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. 2005. Ursa minor: Versatile cluster-based storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, Volume 4 (FAST'05)*. USENIX Association, Berkeley, CA, 5–5.

Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 233–248.

Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. 2001. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM, New York, NY, 43–56. DOI: http://dx.doi.org/10.1145/502034.502040

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. 2006. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev.* 33, 4 (Mar. 2006), 29–35. http://doi.acm.org/10.1145/1138085.1138093

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. 2003. Transforming policies into mechanisms with infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 90–105. DOI: http://dx.doi.org/10.1145/945445.945455

Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*. Retrieved from http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling end-host network functions. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, New York, NY, 493–507. DOI: http://dx.doi.org/10.1145/2785956.2787493

Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, 18–18.

B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, New York, NY, 267–283. DOI: http://dx.doi.org/10.1145/224056.224077

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 143–157. DOI: http://dx.doi.org/10.1145/2043556.2043571

Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. 1996. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.* 14, 4 (Nov. 1996), 311–343. DOI: http://dx.doi.org/10.1145/235543.235544

Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*. ACM, New York, NY, 1–12. DOI: http://dx.doi.org/10.1145/1282380.1282382

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX OSDI*. Retrieved from http://dl.acm.org/citation.cfm?id=1267308.1267323

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings USENIX OSDI*.

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

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. DOI:http://dx.doi.org/10.1145/2408776.2408794

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 205–220. DOI:http://dx.doi.org/10.1145/1294261.1294281

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

Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory networking: An API for application control of SDNs. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. ACM, New York, NY, 327–338. DOI:http://dx.doi.org/10.1145/2486001.2486003

Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, 20.

FreeBSD. 2014. FreeBSD GEOM storage framework. Retrieved from http://www.freebsd.org/doc/handbook/.

Albert Greenberg. 2015. SDN for the Cloud (Sigcomm 2015 Keynote). Retrieved from http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf.

Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*. Retrieved from http://doi.acm.org/10.1145/1592568.1592576

Kieran Harty and David R. Cheriton. 1992. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. ACM, New York, NY, 187–197. DOI:http://dx.doi.org/10.1145/143365.143511

Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Berkeley, CA, 19–19.

Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 167–181. DOI:http://dx.doi.org/10.1145/2517349.2522722

Intel Corporation. 2014. IoMeter Benchmark. Retrieved from http://www.iometer.org/. (2014).

Michael Isard. 2007. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 60–67. DOI:http://dx.doi.org/10.1145/1243418.1243426

Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. ACM, New York, NY, 3–14. DOI:http://dx.doi.org/10.1145/2486001.2486019

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 52–65. DOI:http://dx.doi.org/10.1145/268998.266644

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, 99–112.

Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 351–364.

Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. 1993. Tools for the development of application-specific virtual memory management. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'93)*. ACM, New York, NY, 48–64. DOI:http://dx.doi.org/10.1145/165854.165867

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (Apr. 2010), 35–40. DOI:http://dx.doi.org/10.1145/1773912.1773922

Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. DOI:http://dx.doi.org/10.1145/279227.279229

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of USENIX OSDI (OSDI'12)*.

Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.

Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* 38, 2 (Mar. 2008), 69–74. DOI:http://dx.doi.org/10.1145/1355734.1355746

Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. 2011. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 57–70. DOI:http://dx.doi.org/10.1145/2043556.2043563

Microsoft. 2010. Virtual Hard Disk Performance. Retrieved from http://download.microsoft.com/download/0/7/7/0778C0BB-5281-4390-92CD-EC138A18F2F9/WS08_R2_VHD_Performance_WhitePaper.docx.

Microsoft Corporation. 2014a. Minidrivers, Miniport drivers, and driver pairs. Retrieved from https://msdn.microsoft.com/en-us/library/windows/hardware/hh439643.aspx.

Microsoft Corporation. 2014b. File System Minifilter Drivers (MSDN). Retrieved from https://msdn.microsoft.com/en-us/windows/hardware/drivers/ifs/file-system-minifilter-drivers.

Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008a. Write Off-loading: Practical power management for enterprise storage. *Trans. Storage* 4, 3, Article 10 (Nov. 2008), 23 pages. DOI:http://dx.doi.org/10.1145/1416944.1416949

Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. 2008b. Everest: Scaling down peak loads through I/O Off-loading. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 15–28.

Netronome. 2016. Agilio Server Networking Platform. Retrieved from https://www.netronome.com/products/overview/.

Oracle. 2010. Oracle Solaris ZFS Administration Guide. Retrieved from http://docs.oracle.com/cd/E19253-01/819-5461/index.html.

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 1–16.

Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ, 13–24.

Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. ACM, New York, NY, 27–38. DOI:http://dx.doi.org/10.1145/2486001.2486022

Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies,*

Architectures, and Protocols for Computer Communication (SIGCOMM'12). ACM, New York, NY, 323–334. DOI: http://dx.doi.org/10.1145/2342356.2342427

Microsoft Research. 2014. Microsoft Research Storage Toolkit. Retrieved from https://www.microsoft.com/en-us/research/project/software-defined-stora ge-architectures/.

Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The linux B-tree filesystem. *Trans. Stor.* 9, 3, Article 9 (Aug. 2013). DOI: http://dx.doi.org/10.1145/2501620.2501623

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM, New York, NY, 213–227. DOI: http://dx.doi.org/10.1145/238721.238779

Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*. Helsinki, Finland, 12. DOI: http://dx.doi.org/10.1145/2342356.2342359

Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc.

SNIA. 2007. Exchange server traces. Retrieved from http://iotta.snia.org/traces/130. (2007).

Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. 2015. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15)*. ACM, New York, NY, 174–181. DOI: http://dx.doi.org/10.1145/2806777.2806933

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of ACM SOSP*.

D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of ACM SOSP*.

Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 182–196. DOI: http://dx.doi.org/10.1145/2517349.2522723

Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. 2006. An architecture for internet data transfer. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation, Volume 3 (NSDI'06)*. USENIX Association, Berkeley, CA, 19–19.

Transaction Processing Performance Council. 2014. *TPC Benchmark E - Rev. 1.14.0*. Standard.

Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Santa Clara, CA.

Theodore M. Wong and John Wilkes. 2002. My cache or yours? Making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC'02)*. USENIX Association, Berkeley, CA, 161–175.

Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. 2007. Tesseract: A 4D network control plane. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, 27–27.

Received October 2016; accepted December 2016