sRoute: Treating the Storage Stack Like a Network

Ioan Stefanovici and Bianca Schroeder
University of Toronto

Greg O'Shea
Microsoft Research

Eno Thereska
Confluent and Imperial College London

Abstract

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 paper 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 VMs. This paper shows significant benefits of customized IO routing to data center tenants (e.g., a factor of ten for tail IO latency, more than 60% better throughput for a customized replication protocol and a factor of two in throughput for customized caching).

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 [53]. Furthermore, most IO stacks support the injection of new stages with new functionality using filter drivers common in most OSes [18, 34, 38], or appliances over the network [48].

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 [43]. 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 paper'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 (§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 (§5.2). In a third example, a control application can route requests to per-tenant caches to maintain cache isolation (§5.3).

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 sSwitches to minimize the number of times an IO traverses them, as well as to minimize IO processing times.

sRoute's approach builds on the IOFlow storage architecture [53]. 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 [35]. 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 paper provides a more complete routing abstraction.

This paper 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 our experience in building three control applications using IO routing: tail latency control, replica set control, and file caching control.

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' file system, buffer cache, scheduler, 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

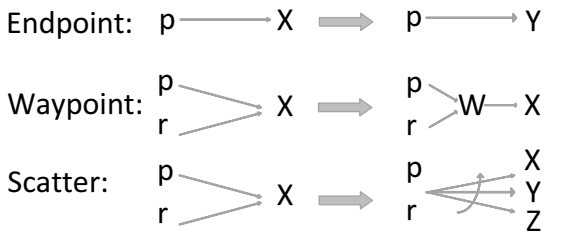


Figure 1: Three types of IO routing: endpoint, waypoint and scatter. p, r refer to sources such as VMs or containers. X, Y, Z are endpoints such as files. W represents a waypoint stage with specialized functionality, for example a file cache or scheduler.

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

Table 1: Examples of specialized functionality and the type of IO routing (E)ndpoint, (W)aypoint and (S)catter that enables 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 paper 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 1 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* [14, 41], *copy-on-write* [21, 42, 46], *file versioning* [37], and *data re-encoding* [1]. These policies have in common the need

for a dynamic mechanism that changes the endpoint of new data and routes IO to the appropriate endpoint. Section §5.1 shows how we implement tail latency control using endpoint 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 [48]. 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 [33, 51] 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.

2.1 Challenges

IO routing is challenging for several reasons:

Consistent system-wide 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, 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 **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 2. 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** 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 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 [53]. 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 paper takes the view that data center services are better positioned for IO stack customization. These are then exposed to applications through new

[1] This paper's implementation uses VMs.

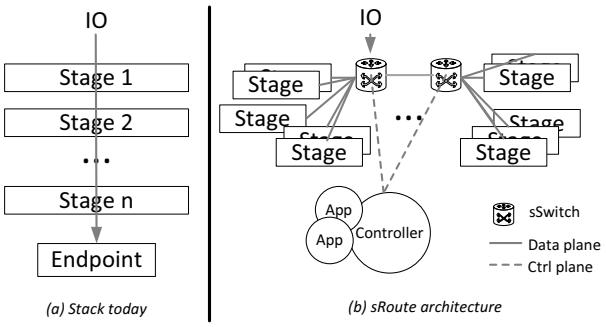


Figure 2: System architecture. sSwitches can route IO within a physical machine's IO stack and across machines over the network.

types of service level agreements (SLA), e.g., 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 5 focuses on control applications. Figure 3 provides the construct definitions used in the rest of the paper.

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 2. 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

[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.

`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

Table 2: Control API to the sSwitch.

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	:=	$< IO \| IOHeader, DetourLoc >$
DetourLoc	:=	$File \| Stage \| Controller$
Stage	:=	$< HostName, DriverName >$
F	:=	Restricted code

Figure 3: Construct definitions.

affected. The source of an IO can be a process or a VM uniquely authenticated through a security identifier. The destination is a file in a (possibly remote) share or directory. Building on IOFlow's classification mechanism [53] allows an sSwitch to have visibility over all the above 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 or just its 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 (§4). `Open` and `create` operations are expensive synchronous metadata operations. There is an inherent tradeoff between lazy file creation upon the first IO arriving and file creation upon rule installation. The former avoids unnecessarily creating files for rules that do not have any IO matching them, but upon 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 paper 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 [53]. `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 of 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 [24].

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 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, //server S_2/C>))
5:     Quiesce (< VM_i, *, //S_1/X>, false)
```

The first two lines are needed to complete any IOs in-flight. This is done so that 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 in line 4 simply returns the cache stage. The controller also injects an sSwitch at server S_2 where the specialized cache resides, so that 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

[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.

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

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 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 paper 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 2 and may not otherwise keep or create any other state. They may insert or delete rules, but may not rewrite the IO header or IO data. That is important since the IO header contains entries such as 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. As part of future work we intend to explore stronger verification of control plane correctness properties, much like similar efforts in networking [27].

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. Similar to networking [45] 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 2 are sufficient to provide per-IO consistent updates.

Per-flow consistency. Many application 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 reading and writing to the same file X, 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 is 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, i.e., reads and writes, to Y, however these sSwitches do not need to perform two-phase commit. The choice between the last two options

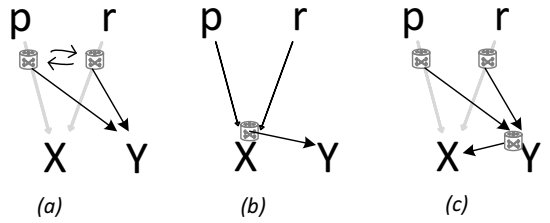


Figure 4: Three possible options for placing sSwitches for consistent rule updates. Either can be chosen programmatically at runtime.

depends on the workload. If the control application expects that most IO will go to the new file 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 continues 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 [31]. If the controller is temporarily unavailable, 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, e.g., using 3-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 which servers IO has been forwarded to 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 limitations

In the course of working with sRoute we have identified several design limitations. First, 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 [27], that we believe could also apply to storage, but we have not investigated them yet.

Second, we now have experience with SDN controllers and SDS controllers like the one in this paper. It would be desirable to have a control plane that understands both the network and storage. For example, it is currently possible to get into inconsistent end-to-end policies when 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.

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 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.

Routing within a server's IO stack. Our implementation makes use of the filter driver architecture in Windows [39]. 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 paper, 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 [39]. 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, using miniport 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, the sSwitches can intercept `lock`/`unlock` calls and maintain the necessary state, however this is not currently implemented.

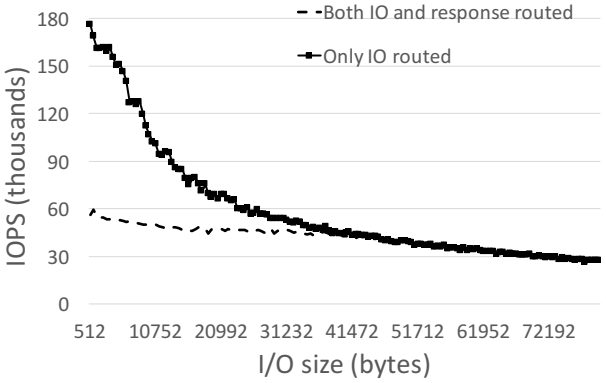


Figure 5: Current performance range of an sSwitch.

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 paper) and the performance difference is a result of the OS, not of the sSwitch. Thus the performance range for small IO is between 50,000-180,000 IOPS which makes sSwitches appropriate for an IO stack that uses disk or SSD backends, but not yet a memory-based stack.

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 hardcoded implementations. We have built and evaluated three control applications implementing tail latency control, replica set control and file cache control. These applications cover each of the detouring types in Table 1. Second, we show that tenants benefit significantly from the IO customization

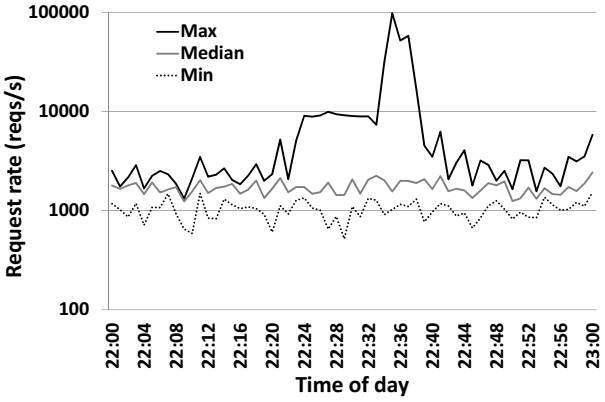


Figure 6: Load on three Exchange server volumes showing load imbalances.

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 2.4 GHz cores, 384 GB of RAM and Seagate Constellation 2 disks. The servers run Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40 Gbps 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 [55] 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 [49]. The third workload is IoMeter [23], 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 [14]. 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 [49], 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 15 minutes.

Data center providers have load balancing solutions for CPU and network traffic [19]. 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 [41] 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 one minute, meant to capture the workload's recent request rate. The controller then temporarily forwards IO if:

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

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 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, 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[6]. The control application in-

[5]The controller uses IOFlow's `getQueueStats` API [53] to gather system-wide statistics for all control applications.

[6]We have also implemented a 1:1 layout that uses sparse files, but do not describe it here due to space restrictions.

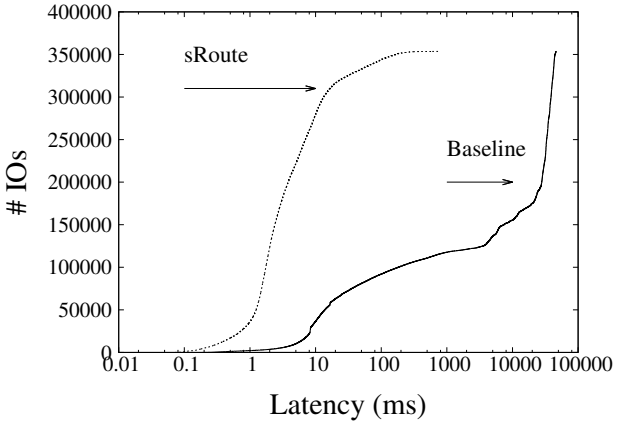


Figure 7: CDF of response time for baseline system and with IO routing.

serts 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 is 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 is 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, the sSwitches and specialized stage removed from the IO stack, since all data resides 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 30 minute segment of the trace, capturing the peak interval and allowing for all forwarded data to be reclaimed. 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. 2.8GB of data was 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 30 seconds to recover the rules from the storage server. The performance benefit obtained is similar to specialized implementations [41]. The CPU overhead at the controller was less than 1%.

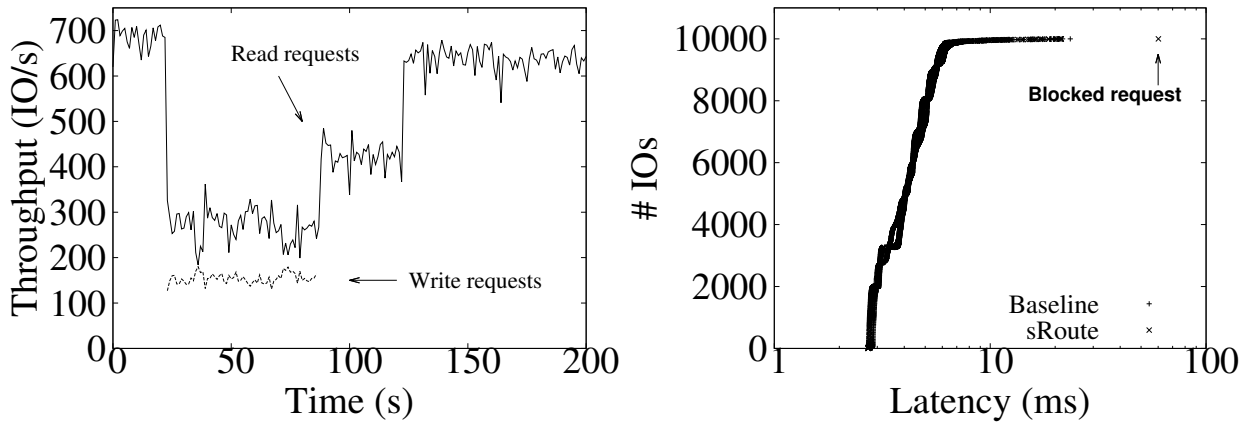
5.2 Replica set control

No one replication protocol fits all workloads [1, 33, 51]. 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., [7]). 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 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, i.e., 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 2-way primary-based replication on the disk testbed. IoMeter uses 4KB random-access requests and each IoMeter maintains 4 requests outstanding (MPL).

The control application monitors the read:write ratio of the workload through IOFlow and when it detects it has been read-only for more than 30 seconds (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 that picks the next server at random. To make the switch between old and new rule the controller firsts `quiesces`



(a) Read and write throughput (b) Write latency CDF

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

writes, then `drains` them. It then inserts the new read-set rule (rule 1):

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(a). The tradeoff is that the first write requests that arrive incurr a latency overhead from being temporarily blocked while the write is signalled to the controller, as shown in Figure 8(b). 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 2x, but the application (IoMeter) has a low MPL and does not saturate storage in this example.

Overheads. The control application changes the forwarding rules infrequently at most every 30 seconds. 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 [2, 8, 16, 50]. The only abstraction the data center provides to a tenant today is a VMs'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 [50] 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 [50] installed in a cache. 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 con-

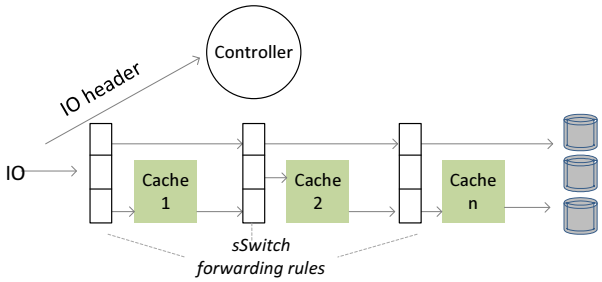


Figure 9: Controller sets path of an IO through multiple cache using forwarding rules in sSwitches.

troller 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.

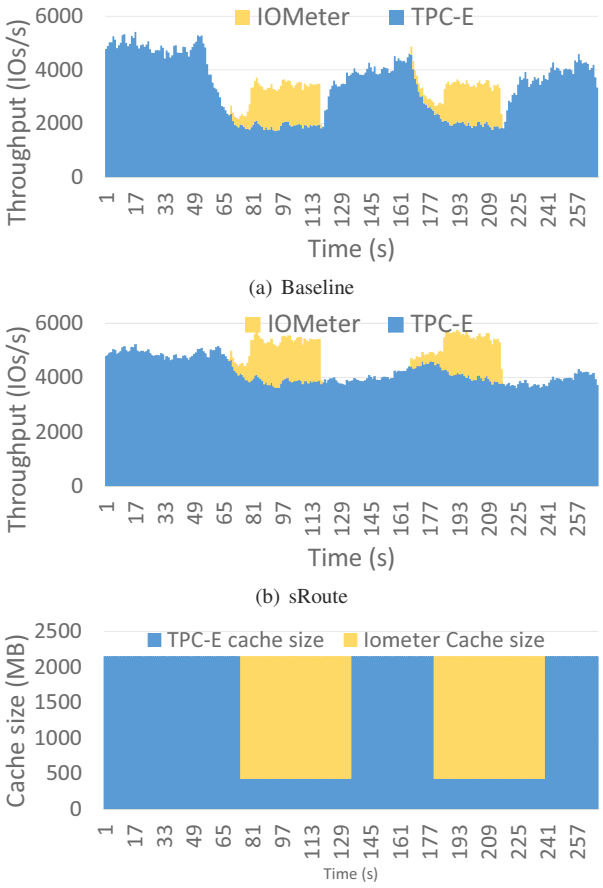
The experiment uses two workloads, TPC-E and IoMeter, competing for a storage server's cache. The storage backend is disks. The TPC-E workload represents queries from an SQL Server database with a footprint of 10GB running within a VM. IoMeter is a random-access read workload with IO sizes of 512KB. sRoute's policy in this example is to maximize the utilization of the cache with the hit rate measured 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 [56].

Then, the controller sets up sSwitches so that the IO 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 2x.

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



(a) Baseline

(b) sRoute

(c) Estimated cache sizes

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

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 15 seconds 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 30 seconds). 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 1 second). The statistics are around 456Bytes/flow (roughly 43MB for 100,000 flows). We believe these are reasonable control plane overheads. When SHARD ran it consumed 100% of two cores at the controller.

6 Open questions

Our initial investigation on treating the storage stack like a network provided useful insights into the pros and cons of our approach. We briefly enumerate several open questions that could make for interesting future work. In Section 3.7 we already discussed two promising areas of future work: 1) sRoute currently lacks verification tools that could help programmers and 2) it would be interesting to merge a typical SDN controller with our storage controller into one global controller.

Related to the first area above, more experience is needed, for example, to show whether sRoute rules from multiple control applications can co-exist in the same system safely. 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 the destination it is considered at rest. It might be advantageous for an sSwitch to initiate itself data movement for such data at rest. That would require new forwarding rule types and make an sSwitch more powerful.

7 Related work

Our work is most related to software-defined networks (SDNs) [9, 13, 17, 25, 28, 44, 54, 58] and software-defined storage (SDS) [2, 53]. Specifically, our work builds directly upon the control-data decoupling enabled by IOFlow [53], 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 paper provides the full routing abstraction.

There has been much work in providing applications with specialized use of system resources [2, 4, 6, 16, 26]. The Exokernel architecture [16, 26] provides applications direct control over resources with minimal kernel involvement. SPIN [6] and Vino [47] 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 [2–4, 36]. 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 paper makes the observation that modern IO stacks support mechanisms for injecting stages with specialized functionality (e.g., in Windows [38], FreeBSD [18] and Linux [34]). 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 a tenants' 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 [8, 20, 22, 29, 32, 50, 57]. 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 [42], WAFL [21] and btrfs [46]. Similarly, Narayanan et al. describe an implementation of load balancing through IO offloading of write requests [40, 41]. Abd-el-malek et al. describe a system implementation where data can be re-encoded and placed on different servers [1]. Finally, several distributed storage systems each offer different consistency guarantees [5, 7, 10–12, 15, 30, 33, 51, 52] .

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 paper 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 paper shows how sRoute can provide unmodified applications with specialized tail latency control, replica set control and achieve file cache isolation, all to substantial benefit.

9 Acknowledgements

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

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of USENIX FAST*, Dec. 2005.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of ACM SOSP*, Banff, Alberta, Canada, 2001.

[3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, L. N. Bairavasundaram, T. E. Denehy, F. I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, Mar. 2006.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with Infokernel. In *Proceedings ACM SOSP*, SOSP '03, Bolton Landing, NY, USA, 2003.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*, Asilomar, California, 2011.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings ACM SOSP*, Copper Mountain, Colorado, USA, 1995.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings ACM SOSP*, Cascais, Portugal, 2011.

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, Nov. 1996.

[9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, 2007.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX OSDI*, Seattle, WA, 2006.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings USENIX OSDI*, Hollywood, CA, USA, 2012.

[13] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: Scalable high-performance storage on virtualized non-volatile memory. In *Proceedings USENIX FAST*, pages 17–31, Santa Clara, CA, 2014.

[14] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM SOSP*, Stevenson, Washington, USA, 2007.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of ACM SOSP*, Copper Mountain, Colorado, USA, 1995.

[17] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking:

An API for application control of SDNs. In *Proceedings of ACM SIGCOMM*, Hong Kong, 2013.

[18] FreeBSD. Freebsd handbook - GEOM: Modular disk transformation framework. http://www.freebsd.org/doc/handbook/.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, Barcelona, Spain, 2009.

[20] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of ACM ASPLOS*, Boston, Massachusetts, USA, 1992.

[21] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX ATC*, San Francisco, California, 1994.

[22] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of ACM SOSP*, Farminton, Pennsylvania, 2013.

[23] Intel Corporation. IoMeter benchmark, 2014. http://www.iometer.org/.

[24] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of ACM SIGCOMM*, Hong Kong, China, 2013.

[26] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of ACM SOSP*, Saint Malo, France, 1997.

[27] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI*, Lombard, IL, 2013.

[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of USENIX OSDI*, Vancouver, BC, Canada, 2010.

[29] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of ACM OOPSLA*, Washington, D.C., USA, 1993.

[30] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[31] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[32] C.-H. Lee, M. C. Chen, and R.-C. Chang. Hipec: High performance external virtual memory caching. In *Proceedings of USENIX OSDI*, Monterey, California, 1994.

[33] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of USENIX OSDI*, OSDI'12, Hollywood, CA, USA, 2012.

[34] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[36] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of ACM SOSP*, pages 57–70, Cascais, Portugal, 2011.

[37] Microsoft. Virtual hard disk performance. http://download.microsoft.com/download/0/7/7/0778C0BB-5281-4390-92CD-EC138A18F2F9/WS08_R2_VHD_Performance_WhitePaper.docx.

[38] Microsoft Corporation. File system minifilter allocated altitudes (MSDN), 2013. http://msdn.microsoft.com/.

[39] Microsoft Corporation. File system minifilter drivers (MSDN), 2014. http://code.msdn.microsoft.com/.

[40] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[41] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings USENIX OSDI*, San Diego, CA, 2008.

[42] Oracle. Oracle Solaris ZFS administration guide. `http://docs.oracle.com/cd/E19253-01/819-5461/index.html`.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[44] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, S. Vyas, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM*, Hong Kong, 2013.

[45] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM*, Helsinki, Finland, 2012.

[46] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[47] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of USENIX OSDI*, Seattle, Washington, USA, 1996.

[48] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*, Helsinki, Finland, 2012.

[49] SNIA. Exchange server traces. `http://iotta.snia.org/traces/130`.

[50] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *ACM Symposium on Cloud Computing (SOCC)*, Kohala Coast, HI, USA, 2015.

[51] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of ACM SOSP*, Farminton, Pennsylvania, 2013.

[52] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of ACM SOSP*, Copper Mountain, Colorado, United States, 1995.

[53] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstrow, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of ACM SOSP*, 2013.

[54] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of USENIX NSDI*, NSDI'06, San Jose, CA, 2006.

[55] TPC Council. TPC-E. `http://www.tpc.org/tpce/`.

[56] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015. USENIX Association.

[57] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of USENIX ATC*, Monterey, California, 2002.

[58] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: a 4D network control plane. In *Proceedings of USENIX NSDI*, Cambridge, MA, 2007.