IOFlow: A Software-Defined Storage Architecture

Eno Thereska[†] Hitesh Ballani[†] Greg O'Shea[†] Thomas Karagiannis[†]
Antony Rowstron[†] Tom Talpey[‡] Richard Black[†] Timothy Zhu[†]

[†]Microsoft Research [‡]Microsoft

Abstract

In data centers, the IO path to storage is long and complex. It comprises many layers or "stages" with opaque interfaces between them. This makes it hard to enforce end-to-end policies that dictate a storage IO flow's performance (e.g., guarantee a tenant's IO bandwidth) and routing (e.g., route an untrusted VM's traffic through a sanitization middlebox). These policies require IO differentiation along the flow path and global visibility at the control plane. We design IOFlow, an architecture that uses a logically centralized control plane to enable high-level flow policies. IOFlow adds a queuing abstraction at data-plane stages and exposes this to the controller. The controller can then translate policies into queuing rules at individual stages. It can also choose among multiple stages for policy enforcement.

We have built the queue and control functionality at two key OS stages– the storage drivers in the hypervisor and the storage server. IOFlow does not require application or VM changes, a key strength for deployability. We have deployed a prototype across a small testbed with a 40 Gbps network and storage devices. We have built control applications that enable a broad class of multi-point flow policies that are hard to achieve today.

1 Introduction

In recent years, two trends have gained prominence in enterprise data centers– virtualization of physical servers and virtualization of storage. Virtual machines (VMs) on the physical servers are presented with a virtual disk. This virtual disk is simply a large file on a shared storage server accessible across the shared data center network. Though these trends have delivered on the promise of reduced costs and easier management, ironically they have also resulted in increased end-to-end complexity. The path of an IO request from a VM to a storage back-end is complex, traversing many layers. A request also appears differently at each layer. For example, a file IO request like a `read`, `write`, `create` in a VM results in a block IO request in the hypervisor. This, in turn, results in Ethernet packets across the network, and finally another file IO request and block device request at the storage server. Requests may traverse many other layers; for the Windows I/O stack there are over 18 stackable layer classes with functionality such as compression, replication, deduplication, virus checking etc. [15]

Such complexity means that enforcing *end-to-end* (e2e) policies is hard. It requires layers along the IO path to treat requests differently based on their contents. Further, policies may need to be enforced at one or more or all layers along the path. For example, prioritizing IOs from a VM to storage requires configuring all layers along the path. This simple policy governs a *point-to-point* IO flow, i.e., all IOs from one endpoint to another endpoint. Policies regarding *multi-point* flows are even harder to enforce. For example, guaranteeing a tenant's IO bandwidth requires dynamic layer configuration across the paths from all its VMs to the storage servers.

This paper proposes IOFlow, an architecture that enables e2e policies in data centers. The policies specify the treatment of IO flows from virtual machines to shared storage. Flows are named using a four-tuple comprising human-friendly high-level identifiers: {*VMs, operations, files, shares*}. For example, if a tenant with 100 VMs that perform data mining on files in "dataset A" is to be given high priority, the policy can be written as {VM 1-100, *, *, "dataset A"} → High Priority.

IOFlow comprises three components. First, a logically centralized controller discovers data plane *stages* (i.e., layers that are IOFlow compliant), and maintains a stage-level data center topology graph. Second, data-plane queues allow for differentiated treatment of IO

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the Owner/Author(s).
SOSP'13, Nov. 3–6, 2013, Farmington, PA, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522723

requests. Stages expose a simple control interface that specifies the low-level identifiers that can be used to direct requests to queues. This interface allows the controller to create queues and dynamically configure their *service* and *routing* properties. For example, by configuring a queue to be serviced at a specified rate, the controller can ensure an IO flow achieves guaranteed performance. Third, we specify a simple interface between the controller and control applications that can be built on top. These applications translate policies into stage-specific configuration disseminated by the controller.

With IOFlow, the controller has global visibility. This allows control applications to use centralized algorithms to translate policies into stage configuration. Such translation includes determining "where" a policy needs to be enforced and "what" queuing rules are needed. By contrast, designing decentralized versions of such applications is harder.

The gains offered by centralized control algorithms have also motivated software-defined networking (SDN) [5, 6, 12, 21, 32]. However, enabling policies for storage IO flows requires classifying and controlling storage IO requests which is difficult to achieve at the network layer (at NICs and switches). IOFlow borrows several SDN ideas and applies them to shared storage, enabling a *software-defined storage architecture (SDS)*. A key challenge was designing queues and rate limiters for storage flows. While network devices have always been able to queue packets based on the network header, configurable queues at storage stages do not exist today; partly because the header for an IO request changes as it traverses across layers. Further, rate limiting is hard because the relation between IO operation and processing time is a non-linear function of IO type, data locality, device type and request size.

We have added control to two key stages in the Windows IO stack: the storage drivers in the hypervisor and the storage server. This allows us to enforce policies with unmodified applications and VMs. We have also added control to other optional stages: a malware scanning device driver, a guest OS file system in the VM, and the network drivers. We have deployed IOFlow on a 12-server testbed of 120 VMs. Through two control applications we built on IOFlow, we illustrate how the system overcomes several challenges: data and control plane efficiency in fast 40 Gbps RDMA-based IO paths, incremental deployability, end-to-end flow name resolution and dynamic control.

2 Scope and challenges

This paper focuses on management of enterprise data centers. Such data centers comprise compute and storage servers. The compute servers are virtualized and host virtual machines (VMs). Each user or *tenant* of the data center is allocated a group of VMs and can run arbitrary applications on its VMs. The storage servers act as front-ends for back-end storage. Storage is usually virtualized, i.e., VMs are often unaware of the details of the interconnect fabric and the storage configuration. VMs are presented with virtual hard disks or VHDs that are simply large files on the storage servers. Such storage virtualization eases management tasks like VM migration and dealing with storage failures.

The compute and storage servers are connected through a network switch that carries both IP traffic between the VMs, and storage traffic from VMs to storage and between the storage servers themselves. While our design can accommodate all these, in this paper we focus only on IO requests from VMs to storage. We also assume the data center has been provisioned appropriately such that the performance bottleneck is at the storage servers; small IO requests are typically interrupt limited while large requests are limited by the bandwidth of the storage back-end or the server's network link.

IOFlow's design targets small-to-medium data centers, with tens of storage servers, hundreds of physical compute servers and 8-16 VMs per server, resulting in O(thousand) VMs. While our ideas can be extended to larger public data centers like Amazon EC2 and Windows Azure, they pose tougher scalability requirements. We defer an exploration of such scaling to future work.

2.1 Example policies

A key challenge in enterprise data centers is enforcement of management policies for storage IO flows. Unlike a network flow which refers to a transport connection between two endpoints, we use the term "**flow**" to refer to all IO requests to which a single policy applies. So flows can be *multi-point* with one or more source endpoints and one or more destination endpoints. The flow endpoints are named using high-level identifiers like VM name for the source, and file name and share name for the destination. For ease of exposition, below we use example flow policies of the form: {[Set of VMs], [Set of storage shares]} $\rightarrow$ Policy. We focus on policies that dictate the performance and routing of flows. Such policies could be specified by data center administrators, management software within the data center or by tenants themselves.

Policy **P1.** $\{$VM p, Share $X\} \rightarrow$ Bandwidth B. VM p runs a SQL client that accesses SQL data files on storage share X. To ensure good query performance, p is guaranteed bandwidth B when accessing the share.[1]

[1] B is in tokens/sec. The relation of tokens to actual IO operations is detailed in Section 3.2.

	Policy	Where to enforce?	What to enforce?
P1	$\{p, X\} \rightarrow B$	C(p) Or S(X)	Static rate limit
P2	$\{p, X\} \rightarrow$ Min B	C(p) Or S(X)	Dynamic rate limit
P3	$\{p, X\} \rightarrow$ Sanitize	C(p) Or S(X)	Static routing
P4	$\{p, X\} \rightarrow$ Priority	C(p) & S(X)	Static priority
P5	$\{[p, q, r], [X, Y]\} \rightarrow B$	C(p), C(q) & C(r) Or S(X) & S(Y)	Dynamic VM Or Server rate limits

Table 1: E2E policies may require distributed and dynamic enforcement. C(p) refers to the compute server hosting VM p, S(X) refers to the storage server where share X is mounted.

Policy **P2.** $\{p, X\} \rightarrow$ Min bandwidth B. Similar to policy P1, but when other VMs are idle, p is allowed to exceed its bandwidth guarantee.

Policy **P3.** $\{p, X\} \rightarrow$ Sanitize. VM p's IO traffic must be routed through a sanitization layer.

Policy **P4.** $\{p, X\} \rightarrow$ High priority. VM p runs a SQL client that accesses SQL log files on storage share X. To ensure low latency for log operations, p's storage traffic requires high priority treatment along the e2e path.

Policy **P5.** $\{[p, q, r], [X, Y]\} \rightarrow$ Bandwidth B. VMs p, q and r belong to the same tenant and when accessing share X and Y, they are guaranteed bandwidth B. Such *per-tenant guarantees* are useful since any of the VMs involved is allowed to use the bandwidth guarantee.

Policies P1–P4 specify treatment of point-to-point flows whereas P5 applies to multi-point flows.

2.2 Challenges

We use the examples above to highlight why IO flow policies are difficult to enforce in today's data centers.

Differentiated treatment. To enforce policies, layers along the flow path need to treat packets differently based on their contents (header and data). For example, consider policy P1. Storage traffic from VM p traverses the guest OS and the hypervisor at the compute server, then the network switch and finally the OS at the storage server before reaching the disk array. To enforce this policy, at least one of these layers needs to be able to control the rate at which requests from VM p to share X are forwarded.

Flow name resolution. Flows are specified using high-level names, e.g., the VM and share name. However, individual layers may not recognize these names, and thus, they may not be able to attribute a request to the flow it belongs to and the policy that applies. For instance, for policy P1, any of the layers from VM p to share X can act as enforcement points, yet each can only observe some low-level identifiers in the requests that traverse them. The flow's destination share X may appear as a file system inside the VM and the guest OS but appears as a block device inside the hypervisor. The hypervisor maps this to a VHD file on the storage server (e.g., "//server/file.VHD"). The storage server, in turn, maps this file (e.g., "H:/file.VHD") to a specific device (e.g., "/device/ssd5"). Hence, flow names need to be consistently resolved into low level identifiers that are accessible to individual layers.

Distributed enforcement. Flow policies may need to be enforced at more than one layer along the flow's path. For example, policy P4 entails VM p's packets should achieve high priority, so it needs to be enforced at all layers along the e2e path. Multi-point policies add another dimension to the need for distributed enforcement. For example, policy P5 requires that the aggregate traffic from VMs p, q and r to shares X and Y be rate limited. This can be enforced either at each of the compute servers hosting these VMs or at the storage servers where the shares are mounted.

Dynamic enforcement. Some policies may require static configuration of layers while others require dynamic configuration. For example, policy P1 in Table 1 requires a static bandwidth limit for VM p's traffic. Static enforcement rules are also sufficient for policies P3 and P4. As a contrast, policy P2 requires that the bandwidth limit for VM p should be adjusted based on the spare system capacity (but should never go below the minimum guarantee). Similarly, multi-point policies like P5 that offer aggregate guarantees also require dynamic enforcement rules.

Admission control. Some policies may not be feasible due to the capacity of the underlying physical resources. For example, to meet a bandwidth guarantee for a VM, the storage back-end should have enough capacity to accommodate the guarantee. The challenge here is to determine if a policy is feasible to achieve.

In summary, enforcing flow policies requires the data plane to support traffic differentiation and global visibility at the control plane. Such visibility allows control algorithms to map flow names into low-level identifiers, to determine if a policy is feasible, to decide where to enforce it and how to dynamically change enforcement rules. This motivated our controller-based design which we describe next.

3 Design

IOFlow is a software-defined storage architecture that enables IO flow policies in multi-tenant data centers. IOFlow requires layers along the IO path to implement a simple control interface with seven API calls. Layers that implement this API are called "stages". A logically centralized controller uses the API to configure stages to make local decisions that enable an end-to-end policy.

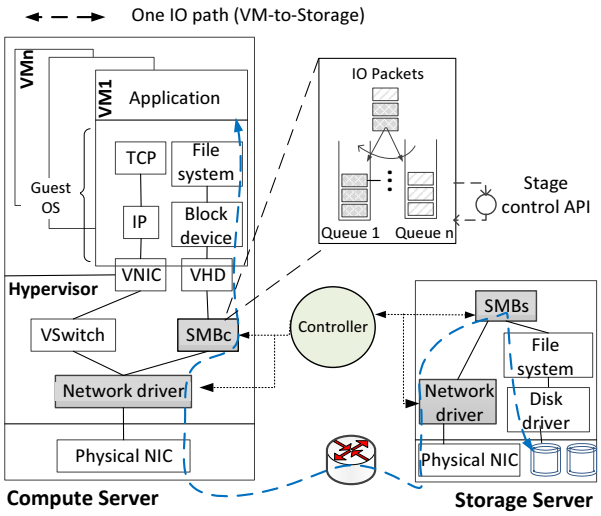

Figure 1: System architecture. IOFlow brings several IO stack stages (shaded) under unified control. Each stage implements queues and queuing rules.

Stages. Figure 1 shows key layers along the IO stack in a typical data center. Any of these can serve as stages. In our prototype, we have implemented the SMB client (SMBc at the hypervisor) and the SMB server (SMBs at the storage server) as stages for enforcing performance policies. SMB is a distributed IO protocol that can run over RDMA. We have also implemented the network drivers as IOFlow stages.

Each stage implements traffic differentiation through queues. Queuing rules map individual IO requests to queues. The stage's control API is shown in Table 2. The API allows the controller to discover the kind of IO headers a stage understands, and can thus use to assign packets to queues (`getQueueInfo`). Further, it allows the controller to create queuing rules (`createQueueRule`) and configure queue properties (`configureQueueService` and `configureQueueRouting`). Queues have service properties that govern how fast they are serviced, and routing properties that dictate the next stage to which IO requests are routed.

Controller. A logically centralized data center controller discovers and interacts with the stages in servers across the data center to maintain a topology graph. It exposes this topology and information about individual stages to *control applications* built on top. These control applications translate high-level flow policies into stage-specific configuration, namely queuing rules for assigning packets to queues and the queue properties. Such translation may be done proactively when the tenant specifies a policy or reactively when a stage receives an IO request that does not match any existing queues and contacts the controller.

A0	**`getQueueInfo`** () returns kind of IO header stage uses for queuing, the queue properties that are configurable, and possible next-hop stages
A1	**`getQueueStats`** (Queue-id q) returns queue statistics
A2	**`createQueueRule`** (IO Header i, Queue-id q) creates queuing rule $i \rightarrow q$
A3	**`removeQueueRule`** (IO Header i, Queue-id q)
A4	**`configureQueueService`** (Queue-id q, <token rate,priority, queue size>)
A5	**`configureQueueRouting`** (Queue-id q, Next-hop stage s)
A6	**`configureTokenBucket`** (Queue-id q, <benchmark-results>)

Table 2: IOFlow's API for data-plane stages.

To show the use of the control API, we focus on Figure 1 and illustrate the enforcement of a simple policy– {VM 4, Share X} $\rightarrow$ Bandwidth B. The controller knows about the following stages on the IO path: SMBc and network driver in the hypervisor, and SMBs and network driver in the storage server. Other layers in the figure are not IOFlow-compliant. No other policies exist and the stages have no queuing rules.

Figure 2 shows the API calls used by the controller at the SMBc stage to enforce this policy. Through call 1, the controller determines the stage understands "File IO" headers. It then creates a queuing rule that directs IOs from VM 4 to server with share X to queue $Q1$. This also causes queue $Q1$ to be created. Note that the File IO header in an IO request contains other fields (like the operation type) that are assumed wildcarded and hence, not matched against. All other IOs are directed to queue $Q0$. Through calls 4 and 5, the controller configures queue $Q1$ to be serviced at rate B while the default queue uses the rest of the storage capacity.

```
1: getQueueInfo (); returns "File IO"
2: createQueueRule (<VM 4, //server X/*>, Q1)
3: createQueueRule (<*, *>, Q0)
4: configureQueueService (Q1, <B,0,1000>)
5: configureQueueService (Q0, <C-B,0,1000>)
```

Figure 2: Controller enforces example policy at SMBc stage. $Q0$ is stage's default queue. C is the capacity of storage back-end.

3.1 Design goals

With IOFlow, we target three main design goals. First, *queues at stages must be fast and cause minimal performance degradation*. Any performance degradation

```
1: IO Header <VM1, //server X/file F> → Queue Q1
2: IO Header <VM2, //server Y/*> → Queue Q2
3: IO Header <VM3, *> → Queue Q4
4: <*, *> → Queue Q3
```

Figure 3: Example SMBc stage queuing rules. "Server" could be a remote machine or the local host.

```
1: IO Header <SID S1, H:/File F> → Queue Q1
2: IO Header <SID S2, H:/File G> → Queue Q1
3: IO Header <SID S2, H:/Directory A/*> → Queue Q2
4: <*, *> → Queue Q3
```

Figure 4: Example SMBs queuing rules. SID stands for security descriptor that identifies VM.

along the IO stack will show up with fast 40+ Gbps RDMA-capable networks and storage devices such as SSDs or in-memory storage like memcached [18] or RamCloud [19]. Second, *the control plane must be flexible, responsive, accurate, resilient and scalable*. The control plane should be flexible, responsive and accurate to allow for rich control application policies. It must survive failures and any temporary unavailability of the control plane should not stall the data plane. It must also be scalable. Third, it is our goal *not to require any application or VM changes*. As we show in this paper, it is possible to implement a rich set of policies benefiting unmodified applications and VMs.

3.2 Stages

Each stage has queues and queuing rules. Queues are the mechanism to provide differentiated flow control. All queues and queuing rules are soft state, i.e., they do not need to survive stage failures. Each stage inspects an incoming IO request, matches it to a queuing rule and forwards it to the appropriate queue. Queuing rules are checked in the order they were created with the default rule, if one exists, being checked last. If no match exists, the request is blocked while the stage requests a queuing rule from the controller that is subsequently installed at the stage. Figure 3 shows example queuing rules at the SMBc stage in the hypervisor; Figure 4 shows them for the SMBs stage in the storage server. Note that stages can use different low level IO Headers for queuing packets. The controller resolves a flow name to queuing rules for individual stages. Also, different queuing rules can refer to the same queue as in Rules 1 and 2 in Figure 4.

Stage queues have two main properties: *service* and *routing*. A stage may allow the controller to configure one or both or none of these through its control interface.

Service properties. A stage that implements queues with configurable service properties can throttle or treat IO requests preferentially. Such service properties are needed for performance isolation policies (e.g., policies P1,P2,P4,P5). To throttle IO traffic, queues use a token-bucket abstraction [31]. The queues are served at `token rate`. Some queues can be treated preferentially to others as indicated by the `priority` field. If a queue reaches the `queue size` threshold, the stage notifies the controller and blocks any further inserts to that queue. There is no support in the IO stack for dropping IO requests, but there is support for blocking requests and applying back-pressure. The controller can set the service properties (<`token rate`, `priority`, `queue size`>) for a queue using the `configureQueueService` call. Periodically it can use `getQueueStats` to monitor statistics on the queue, i.e., its average service rate and queue size.

Queue routing. Some stages may allow control over how IO requests are routed. Queues are associated with a default next-hop. For example, requests at the SMBc stage are routed to the network driver. It may allow requests to be routed to a different stage, perhaps a stage not even in the hypervisor. Such configurable plumbing of stages can allow for a rich set of flow policies. For example, Section 4.2 shows how the controller uses the `configureQueueRouting` call to route IO requests from untrusted VMs through a malware scanner stage.

Storage request peculiarities. Storage requests are different from network packets, so the design of the token bucket is different for them too. For network packets, a single byte is represented by a single token. Storage requests pose three challenges. First, they represent different operations like `read`, `write` and `create`. While a `write` operation contains the payload, a `read` operation starts small in length (usually just a header) at the sender and the response contains the payload. Thus, at sender stages, instead of releasing tokens based on bytes in the request, they need to be released based on the end-to-end cost of the operation.

Second, operation processing time at the storage back-end is also variable. Operation type, request size, data locality and device type all impact processing time. Thus, unlike network packets, the relation between storage operation and tokens can be a non-linear function of the above properties. To address this, the controller's discovery component benchmarks the storage devices to measure the cost of IO requests as a function of their type and size (see §3.3). It then uses the `configureTokenBucket` call to configure stages with information regarding the number of tokens to be charged for any given request. Such configuration is done periodically since the cost of IO requests varies with varying aggregate workload to the storage device. While this simple approach works well in our experi-

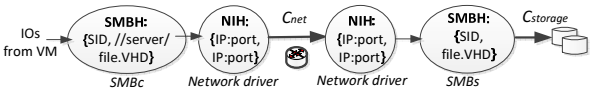

Figure 5: The topology graph corresponding to the physical setup in Figure 1. The nodes are stages that implement IOFlow's control interface. C_{net} and $C_{storage}$ are the capacities of the physical resources.

ments, we note that accurately determining the cost of IOs with varying workloads is an active research problem; for example, mClock [9] uses an analytical expression for the cost of IOs against mechanical disks.

Third, to bound the performance uncertainty of arbitrary long IO requests, we can instruct SMB to split them; e.g., a 10 MB request can be split into 10 requests of 1 MB with minimal performance penalty [29].

Stage efficiency. Data plane queues need to be very efficient to handle high speed 40+ Gbps networks and storage. By having global visibility, IOFlow's controller chooses *where* to implement the policy so as to minimize performance impact. This is in addition to standard techniques such as zero-copying of requests moving from stage to stage and efficient min-heap-based selection of which queue to service next within a stage. For example, for a tenant with VMs across 10 hypervisors accessing the same storage server, the controller prefers to do rate limiting at each of the hypervisors for greater parallelism, rather than at the storage server.

3.3 Controller

The controller discovers stages in the data center, provides control applications with information needed to implement e2e policies by converting them to stage-specific configuration and then disseminates this configuration to the stages.

3.3.1 Discovery component

The controller's discovery component maintains a stage-level graph for the data center. When servers boot up and device drivers are initialized, stages contact the controller whose location is found through existing data center management interfaces. Figure 5 shows a graph with four stages corresponding to the setup in Figure 1. It also includes the underlying physical resources, i.e., the network link, the network switch and the storage back-end. A simple example IO flow is shown, with a VM accessing a VHD. The discovery component uses API call `getQueueInfo` to get information about the stages. This includes the low-level IO header a stage can use to assign packets to queues: SMB headers (SMBH) and network IO headers (NIH).

The discovery component also determines the capacity of the graph edges corresponding to physical resources. For edges representing physical network links, it relies on a pre-configured data center topology map. For edges to the storage back-end, it runs a series of benchmarks based on IoMeter [10] where IO packet size, type, and locality are parameterized to cover a wide range of workload characteristics. This benchmarking is fully parallelizable, and only needs to be re-run in case of hardware changes.

Flow name resolution. Flows are named using high-level identifiers while stages can only observe IO headers in packets. Thus, the controller needs to resolve flow names into stage-specific queuing rules. For example, since the controller knows the SMBc stage at the hypervisor understands SMB headers, it generates queuing rules of the form shown in Figure 3.

3.3.2 Churn, updates and ordering

The controller may need to update several stages when a new policy is defined or updated, when the location of existing VMs or shares changes, or if there are failures. The controller deletes queuing rules at the stages along the old flow path and adds queuing rules along the new flow path.

There are policies that do not require any particular update ordering *across* stages and can tolerate temporary inconsistent rules at each stage as long as they eventually converge. Performance might slightly degrade during such inconsistencies. For this set of applications, the controller simply batches any state updates with a version number to the relevant stages, waits for acks with the same version number and only then proceeds with any further state dissemination. If control applications require strict update ordering the controller updates each stage in the needed order without requiring stages to participate in distributed agreement protocols.

Each stage's configuration is soft state. Failure of any stage along a path (e.g., storage server failure) will destroy all queues and queuing rules on that stage. When the server comes back online, it needs to contact the control service to get a list of queuing rules it is responsible for. No request is processed until that completes. Hence, the time to repopulate the soft state adds to the period of unavailability for that storage path. Server reboot time, however, still dominates the unavailability time.

The controller keeps track of a *version* number associated with configuration messages to stages, and monotonically increments that number each time it contacts the stages. Upon receiving a message with a version number, the stages discard any subsequent messages with lower version numbers (that could have been in the system after a temporary network partition).

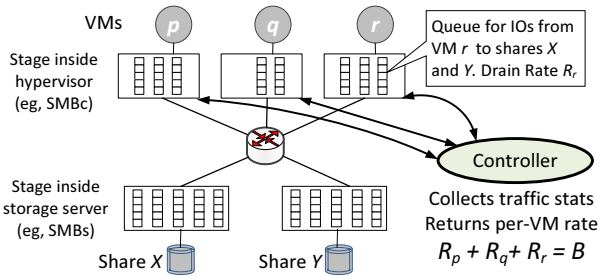

Figure 6: Topology sub-graph comprising stages relevant for enforcement of policy 5 (other stages along the path from VMs to storage are omitted for clarity).

Request ordering. IOFlow does not change application request ordering semantics. IOFlow can delay IOs or divert them to different queues. However, applications already ensure ordering semantics at a higher level in order to deal with, for example, resources like disks that could complete requests in any order. Applications have two ways of ensuring ordering: either use synchronous IO or use asynchronous IO with explicit `fsync` calls. IOFlow does not change the semantics in either case.

4 Control applications

We have built two control applications on IOFlow. The first application enables performance policies like P1, P2, P4 and P5. We describe it in depth since it provides context for general control applications. The second application enables control over flow routing (policy P3).

4.1 Performance control

This control application enables three broad classes of performance policies for both point-to-point and multi-point flows. For each IO flow, a bandwidth guarantee or a minimum bandwidth guarantee can be specified. With the former, bandwidth is reserved for the flow and can not be exceeded. Further, for each flow, its priority can be specified. This priority will be enforced at all stages along the flow path, and supports applications that require low IO latency.

To illustrate the operation of this control application, we use policy P5, $\{\{p,q,r\}, \{\text{X, Y}\}\} \rightarrow$ Bandwidth B, as an example. This policy is particularly interesting as it involves a multi-point flow, i.e., it offers an **aggregate guarantee** for IOs from multiple sources (VMs p,q,r) to multiple destinations (shares X,Y). For example, if VM p is the only VM generating IOs, it should achieve bandwidth B. As we describe below, this requires dynamic configuration of stages.

This policy is enforced as follows. Given the data center topology graph maintained by the controller, the control application determines the sub-graph relevant for this policy. As shown in Figure 6, this sub-graph comprises all the stages along the path from VMs p,q,r to the storage servers where shares X and Y are mounted. The edges of this graph are the links between the stages, the physical network links and the interface from the storage servers to the actual disks. The operation of the control application comprises two steps.

Admission control. The controller needs to ensure that all edges along the graph above have enough spare capacity to meet the guarantee. The total bandwidth capacity of the network links and the links to storage is determined by the controller's discovery component. Since some of the capacity on each link may have been reserved for pre-existing policies, this policy can only be admitted if the unreserved capacity on each link is no less than the guarantee B.

Algorithm 4.1 Controller-based distributed rate limiting

Require: N VMs with aggregate guarantee B; D: set of VM demands sorted in ascending order ($D_i > 0$); VM i's IOs are queued at q_i

Ensure: Assign rate R_i to VM i s.t. $\sum R_i = B$

```
1:  leftB = B    // bandwidth left to distribute
2:  for i in [0, N - 1] do
3:      if D_i ≤ leftB/(N−i) then
4:          R_i = D_i        // VM demand is less than fair share
5:      else
6:          R_i = leftB/(N−i)    // VM demand is more than fair share
7:      leftB -= R_i

8:  {share any left bandwidth and configure queues}
9:  for i in [0, N - 1] do
10:     R_i += leftB / N
11:     configureQueueService(q_i, < R_i, 0, 1000 >)
```

Enforcement. The controller needs to ensure that the total rate of all IOs from VMs p,q,r to shares X,Y does not exceed B. This aggregate bandwidth limit can be enforced at any cut in the graph. As shown in Figure 6, it can be enforced at the hypervisors where the VMs are hosted or at the storage servers. For this example, we assume the controller chooses to enforce the bandwidth limit at a stage inside the hypervisors. To do this, it needs to determine the per-VM bandwidth limit such that the sum of the limits is equal to B. Thus, the controller application needs to do *distributed rate limiting* (DRL)– given a set of N VMs (in this example N=3), distribute bandwidth B between them. In contrast to past approaches [22], the presence of a controller with global visibility allows us to achieve DRL through a simple centralized algorithm, as shown in Algorithm 4.1.

This distributed rate limiting problem is akin to dividing a single link of capacity B between N competing sources. A well accepted approach to fairly share the link is to give each source their *max-min fair share* [4]. Max-min sharing assigns rates to VMs based on their traffic demand, i.e., the rate at which they can send traffic. Specifically, VMs are allocated rates in order of increasing demands such that no VM gets a rate larger than its demand and VMs with unsatisfied demands get equal rates.[2] Thus, given the demand set D for the VMs, we determine the max-min fair share f such that when VMs are allocated rates R_i as follows, it ensures $\sum R_i = B$:

$$R_i = min(f, D_i)$$

To enforce the policy, the control application configures the hypervisor stages as follows. For VM p, it creates a queue at a stage in the hypervisor where p is hosted, creates a queuing rule that directs all traffic from VM p to shares X and Y to this queue (through flow resolution and API call `createQueueRule`, not shown in the algorithm) and configures the queue's token rate to the rate allocated to p (line 11). The stage configuration for VMs q and r is similar.

To estimate VM demand, the controller periodically gathers statistics from the hypervisor stages using `getQueueStats`. Every interval, the estimated demand for a VM whose actual IO rate equals its previous rate allocation is set to the aggregate limit B, else the VM's estimated demand is the same as its IO rate. For idle VMs with no IOs, the estimated demand is set to a low value so that they do not get rate limited to zero and can ramp up when needed. Based on the updated demand vector, the control application periodically generates new rate limits for the VMs and updates the token rate for the appropriate queues. This exemplifies how guaranteeing aggregate bandwidth for multi-point flows requires dynamic enforcement.

Min-guarantee. The algorithm above ensures an aggregate guarantee for multi-point flows. Instead, offering an aggregate *minimum* bandwidth guarantee means that when VMs for one tenant are inactive, VMs of other tenants can utilize the spare bandwidth. Ensuring this is conceptually similar. The controller collects traffic statistics from stage queues, and uses a hierarchical max-min sharing algorithm across all data center VMs, instead of IOs belonging to a single tenant, to determine their bandwidth limits. However, we omit the details of this algorithm for brevity.

Priority. Priority at each stage is set using the `configureQueueService` call. When a queue has no outstanding tokens, it is not served. When tokens become available, the highest-priority queues are serviced first, until they run out of tokens. After that, the lower priority queues are serviced next.

[2]While we describe unweighted max-min sharing here, the algorithm can account for weights.

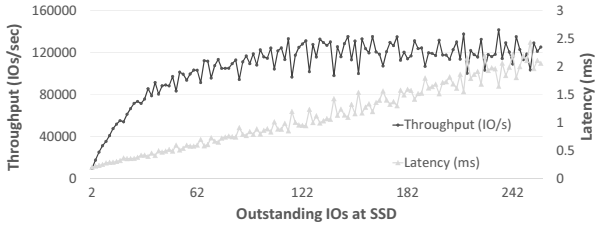

Figure 7: Number of outstanding IO requests vs. throughput and latency for an SSD array (Storage type A in Table 3). The workload is IoMeter (Section 6) using 8KB read requests.

Enforcing end-to-end priority requires support from all stages that see contention for resources. For incremental deployment, IOFlow's API allows the system to tolerate layers that do not implement the control interface by controlling the number of IO requests outstanding in those layers. For example, in our implementation IOFlow does not have control over requests once they enter an SSD. The discovery component runs benchmarks to measure the tradeoff between the token rate, number of outstanding IO requests, and latency for the device. As an example, to keep the SSD (Storage type A in Table 3) 95% utilized, 90 outstanding requests could be sufficient as shown in Figure 7 (other utilization-latency tradeoff points could also be chosen). Thus, IOFlow could control the token rate to maintain 90 requests at the device and the rest in IOFlow's data-plane queues. Priority treatment can then be applied to those data-plane queues.

4.2 Malware scanning middlebox

IOFlow allows runtime control over which IO requests are routed through a specific processing middlebox. As an example, we have implemented the malware scanning control application which enforces policies like P3. The intuition behind this control is that not all flows should be (un)trusted equally. We have modified a standard malware scanner kernel driver template [16] to operate as an IOFlow stage. The default policy is to *always* scan during an `open`, `close` and `write` operation.

The controller routes IO requests from untrusted VMs through the scanning stage using call `configureQueueRouting`. Other requests are routed around this stage. The scanning stage is an optional stage and can reside at either hypervisor or storage server. We verified the policies worked by sending a mix of IOs from both trusted and untrusted VMs, and observing which ones were scanned.

4.3 Impact of controller failure

The control applications are designed so that transient controller failures do not impact system correctness. Failures can lead to temporary degraded performance however. Each control application implements a conservative default policy in case the controller is unreachable. For example, when the controller is unreachable, the performance control application requires that stages put all new traffic for which queuing rules do not exist into a best-effort queue, until the controller is available again to specify their policy. The malware scanning application's default policy is to scan all traffic.

5 Implementation

We implemented IOFlow on a Windows-based IO stack. We implemented two kernel drivers that intercept storage IO traffic and each serves as an IOFlow stage: a storage driver that resides on top of the SMBc driver, and a storage driver below the SMBs driver (see Figure 1). Unmodified binaries and applications can make use of IOFlow's functionality. We rely on the Windows storage IO and network IO stacks for supporting the injection of our storage and network drivers.

We have also added similar control to other optional stages along the IO path: a malware scanning device driver (benefits described in §4.2); a driver on top of the guest OS file system (NTFS) that allows for IO flow differentiation within a VM; network drivers in the hypervisor that allow for control of VM-to-VM traffic.

Stage queue rules are stored as soft-state in each interception driver. The size of the token bucket associated with each queue is configurable; we used a size of `token rate` x 1 sec tokens for our experiments. Tokens are replenished using a 10 ms timer. Each stage communicates with the control service by passing messages to a user-level slave process on the local machine, which then transmits the messages to the controller using RPCs over TCP. The default control interval for the controller is 1 second. For queue routing, a stage's next hop can be any stage on that stage's physical machine, including stages in the kernel and user-level. Routing to a next hop on a remote machine can be done by first routing to user-level and then sending an RPC to the remote machine's user-level, however implementing this feature is future work.

Our system assumes that users and machines are authenticated. In the current system, Windows assigns each VM a unique security descriptor (SID). The SID is a variable length structure, part of the initial `open` or `create` IO request. SIDs can also be assigned to users, not just VMs. The optional driver in the guest OS is able to differentiate traffic based on user SIDs.

Network	Storage A	Storage B	Storage C
1 NIC/server	8 **SSDs**	**RAM**	3 **Disks**
	1.7 TB	384 GB	2.7 TB
40 Gbps	2.7 GB/s*	5 GB/s†	0.3 GB/s
(=5 GB/s)	140K IOs/s*	460K IOs/s†	1K IOs/s

Table 3: Baseline device characteristics. GB/s measured with 512 KB streaming read requests. IOs/s measured with 4 KB random-access read requests. (*) The SSDs' write throughput is 1.5 GB/s and 50K IOs/s. (†) RAM is accessed over the network.

	Index	Data	Message	Log
Read %	75%	61%	56%	1%
IO Sizes	4/64 KB	8 KB	4/64 KB	0.5/64 KB
Seq/rand	Mixed	Rand	Rand	Seq
# IOs	32M	158M	36M	54M

Table 4: Workload characteristics for 4 tenants, part of a 2-day Hotmail IO trace. Seq/rand refer to sequential and random-access respectively. *M*=million.

When changes to the guest OS are acceptable, richer user-based policies can thus be enabled.

Stages along the stack sometimes re-write this SID in-place. For example, the hypervisor converts a VM's IO header SID into a hypervisor SID and passes that to the storage server. Thus, the ability of the storage server to identify which VM triggered the IO is normally lost. To differentiate per-VM traffic at the storage server, we have implemented a small modification in the SMBc stage. Each time SMBc sees an `open` or `create`, it sends an `IOCTL` with the VM's SID as the payload to the SMBs stage on the storage server. SMBs then caches that SID as part of the file handle context.

The kernel drivers are written in C and are around 22 kLOC in total. The controller is written in C# and is around 3 kLOC lines of code. Currently, we run the control service on just one of our 12 testbed servers. It can be replicated for availability using standard techniques.

6 Evaluation

We evaluate IOFlow across two dimensions: i) its ability to enforce e2e policies and ii) the performance and scalability of the control and data plane mechanisms. The evaluation is driven by IO traces of the Hotmail service and the IoMeter file system benchmark [10].

Our testbed comprises 12 servers, each with 16 Intel Xeon 2.4 GHz E5-2665 cores and 384 GB of RAM. Each server has a 40 Gbps RDMA-capable Mellanox ConnectX-3 NIC with a full-duplex port, connected to a Mellanox MSX1036B-1SFR switch. Hypervisors communicate with storage servers using the SMB 3.0 file

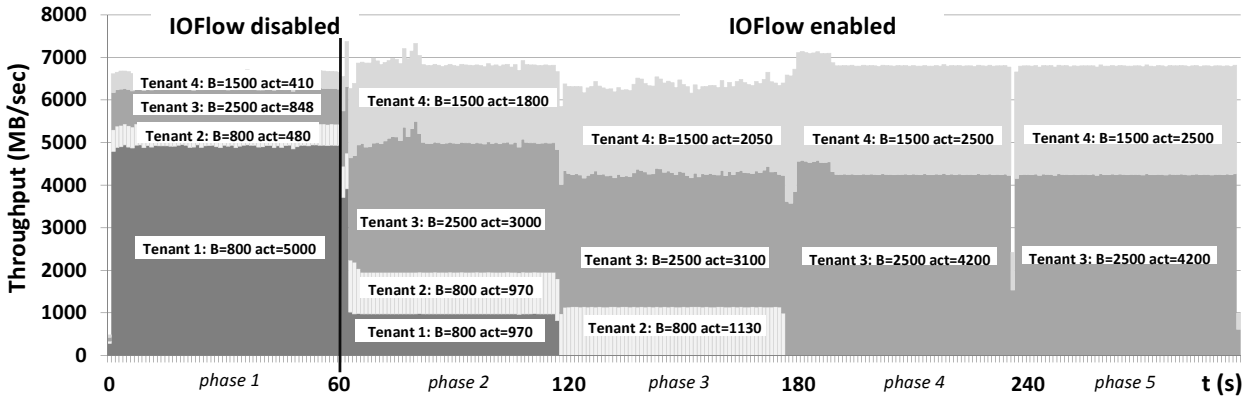

Figure 8: 4 tenants using 120 VMs in total across 10 hypervisors with policies in Table 5. When IOFlow is disabled tenant policies B are not met. With IOFlow enabled tenant policies are met ("actual" $\geqq B$). Furthermore, extra capacity is assigned in proportion to the tenant minimum bandwidth.

server protocol over the SMB Direct RDMA transport. In the link layer we use RDMA over converged ethernet (RoCE). The servers run Windows Server 2012 R2 operating system with Hyper-V as the hypervisor.

Each server can act as either a hypervisor with up to 16 VMs on one hypervisor or as a storage server with three types of configuration as seen in Table 3. Type A uses 8 SSDs (Intel 520) in RAID-0, type B uses RAM only (representative of a cached workload and used to stress-test the system) and type C uses 3 disks (Seagate Constallation 2) in RAID-0. When SSDs or disks are used there is no data caching in RAM.

6.1 Policy enforcement

IOFlow enables all policies discussed in Section 2.1. Here we show that IOFlow allows tenants with diverse policies to co-exist. Our experiments are driven by a set of I/O traces from the Hotmail service [26]. The traces contain four distinct workloads with key characteristics summarized in Table 4. The "Message" workload stores email content in files; 'Index" is a background maintenance activity scheduled at night time in the data center; the "Data" and "Log" workloads are database data and transaction logs respectively. Metadata on emails is stored on these databases.

Bandwidth policies. We first start by examining whether IOFlow is able to enforce *per-tenant minimum bandwidth guarantees*. We treat each of the four workloads as a tenant with its own policy. The exact policies are shown in Table 5. These are multi-point policies combining policy P2 (minimum bandwidth guarantee) and P5 (per-tenant bandwidth guarantee).

We use a total of 120 VMs spread over 10 hypervisors and accessing a RAM-based share X. Each tenant is assigned 30 VMs, spread equally over the 10 hypervisor machines. In this experiment, each tenant's VM uses IoMeter parametrized to match the workload characteristics of one of the Hotmail workloads in Table 4.[3]

Tenant	Policy
1. *Index*	$\{VM1-30, X\} \rightarrow$ Min 800 MB/s
2. *Data*	$\{VM31-60, X\} \rightarrow$ Min 800 MB/s
3. *Message*	$\{VM61-90, X\} \rightarrow$ Min 2500 MB/s
4. *Log*	$\{VM91-120, X\} \rightarrow$ Min 1500 MB/s

Table 5: E2E policies for four tenants accessing a share X. IoMeter is parametrized with workload characteristics from the Hotmail trace.

Figure 8 shows cumulative results for the absolute throughput achieved per tenant and in aggregate. The experiment is separated across five phases, one every 60 seconds. In the first, IOFlow is not running; the minimum guarantee for three of the tenants is not met as a result of Tenant 1 aggressively consuming system resources. Aggressiveness is induced by setting the Tenant 1's IoMeter outstanding requests to 32 per VM; the other tenants use 8 per VM.

Enabling IOFlow at time $t = 60secs$ ensures tenants get the minimum bandwidth specified by their policy. Since the overall capacity is higher than the sum of the guarantees, each tenant receives extra capacity. Extra capacity is assigned in proportion to the tenant minimum bandwidth. At phases 3 ($t = 120secs$) and 4 ($t = 180secs$), tenants 1 and 2 become idle respectively. The controller realizes that extra capacity is available and apportions it *across* active tenants; again, extra capacity is

[3]The traces we obtained are open-loop and do not come close to saturating the bandwidth of our system, hence for this experiment we use closed-loop IoMeter. We will use trace replay of the real workloads in the following experiments where we examine the co-existence of bandwidth and priority policies.

Tenant	Policy
1. *Message*	$\{VM1-4, Y\} \rightarrow$ High priority
2. *Index*	$\{VM5-8, Y\} \rightarrow$ B=16 MB/s

Table 6: E2E policies for IO trace replay.

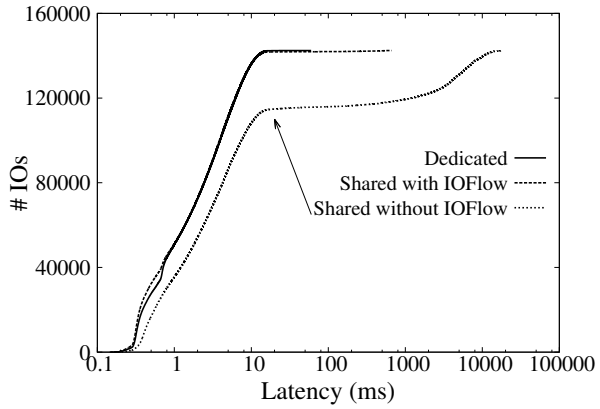

Figure 9: "Message" latency CDFs. The mean and 99^{th} percentile with IOFlow are 4.3 ms and 13.5 ms respectively. Without IOFlow they are 926 ms and 11668 ms respectively. With dedicated hardware they are 3.1 ms and 13 ms respectively.

shared proportionally to tenant guarantees.[4] In phase 5 ($t = 240secs$), half of the VMs of each of the remaining two tenants cease generating work. The controller apportions the extra capacity to the other VMs *within* each tenant. Overall, this example shows that IOFlow efficiently enables multi-point policies, with the controller able to dynamically re-allocate resources based on inter- and intra-tenant work-conservation.

Bandwidth and priority policies. We now turn our attention to workloads with different requirements, namely throughput and latency. To this end, we use a trace replayer to replay the "Message" together with the "Index" workloads from the Hotmail trace on the SSD store. "Message" is a latency sensitive workload, while "Index" is a typical bandwidth-hungry background task. Current practice usually has such workloads separated in time. In our traces, "Index" was scheduled during night time. Here, we will highlight how IOFlow enables such workloads to co-exist.

We replay the "Index" workload in closed-loop fashion ignoring the original timestamps. We believe that this is a reasonable adjustment since maintenance activities are usually triggered by a script and require batch processing. The goal is to complete the "Index" activity within 24 hours. This leads to its policy– a steady state guaranteed IO rate of around 16 MB/s as shown in Table 6. The policy specified for the "Message" workload is high priority, enforced at both SMBc and SMBs stages. Note that, in general, a workload with high priority could starve lower priority workloads. Since the controller has global visibility, it can avert this problem by specifying a bandwidth limit to the high priority workload. In our setup the "Message" workload has an average rate of 62 MB/s and the SSDs can support both workloads' rates so a bandwidth limit is not necessary.

To show worst-case performance, we choose a 10-minute trace interval with the highest 99% arrival rate for the "Message" workload. We load-balance the trace replay into 4 VMs and 2 servers (2 VMs/server) for the "Message" workload and 4 VMs and 2 servers (2 VMs/server) for the "Index" workload. Each VM uses a distinct 127 GB VHD. All 8 VHD files are on the same SSD-based share Y.

Figure 9 shows the results. The latency for the "Message" workload suffers when IOFlow is disabled. Instead, when IOFlow is enabled, the latency is almost identical as to when "Message" has dedicated resources. This shows that IOFlow provides good workload isolation. The "Index" tenant's average bandwidth over the 10-min interval is 17 MB/s.

6.2 Performance and scalability

IOFlow introduces mechanisms both at the control and data planes. This section measures the performance and scalability along the data and control planes.

6.2.1 Data plane overheads

Programmable data plane queues allow for differentiated traffic treatment and workload isolation. This section quantifies their overheads. We vary the IO size with IoMeter to span the range between 0.5 KB and 64 KB and we examine the system throughput achieved. The read:write ratio is kept at 1:1 and IoMeter uses random-access requests. We use the same 120 VMs over 10 hypervisors setup accessing a single storage server. Each VM corresponds to one tenant in this case. The tenant's SLA is a guarantee for $1/120^{th}$ of the network and storage capacity. Requests flow through both SMBc and SMBs stages, but policy enforcement (i.e., rate limiting) is done at the hypervisors' SMBc stages. We vary the storage server configuration to use RAM, SSDs or Disks.

Figure 10 shows the results. For the RAM store, the bottleneck shifts from being the server's CPU for IO sizes up to 16 KB to the network for larger IOs. The worst-case reduction in average throughput between the original system and IOFlow is 14%. For the SSD and disk stores the bottleneck is always the storage device. The worst-case reduction in average throughput was 9%

[4]The slight reduction in throughput during phase 3 is due to the change in workload; the overall read:write ratio changes and Tenant 2's workload (with smaller IO sizes) gets more of the system's resources.

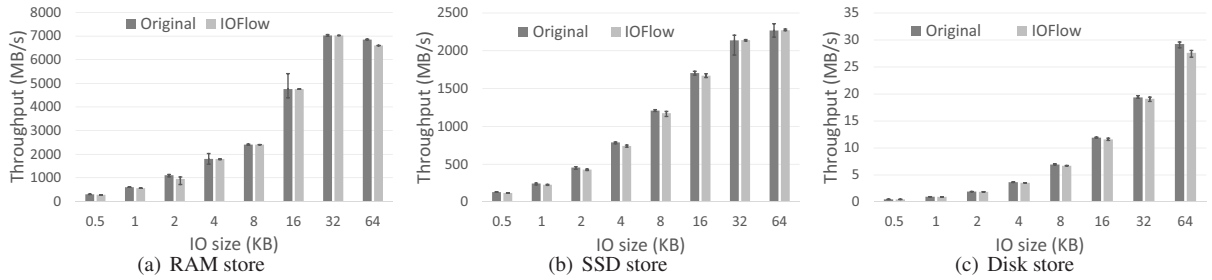

Figure 10: Performance overheads of IOFlow when compared to unmodified storage stack. Error bars show minimum and maximum values from 5 runs. Note that y-axis is different for each graph.

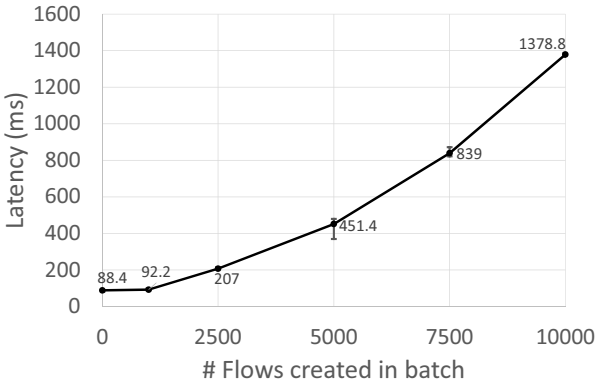

Figure 11: Average flow creation latency. Error bars show minimum and maximum values from 5 runs.

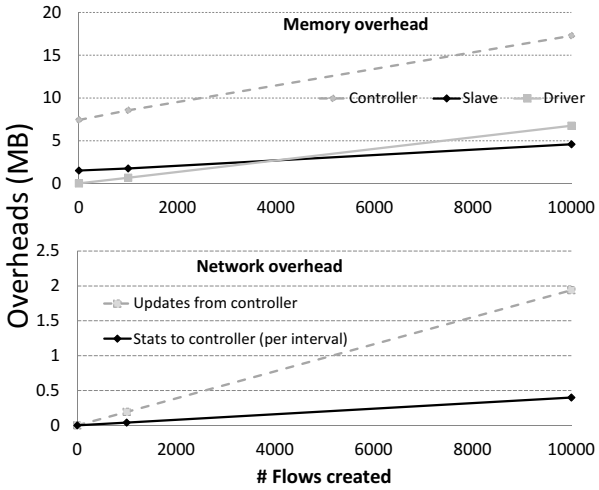

Figure 12: Memory and network overheads associated with creating flows and getting flow statistics.

and 5% respectively. This reduction does not come from any CPU overheads. Instead, it comes from a slight reduction in parallelism at the SSD or disk. This reduction happens because with IOFlow the queues are drained in a certain order. Without IOFlow the request mix is sent directly to the device. Thus, the price to pay for workload isolation is a slight drop in overall throughput. Across all devices, the worst-case overhead in average CPU consumption at the hypervisors is less than 5% (not shown in Figure 10).

6.2.2 Control plane overheads

When a policy for an IO flow is specified, the controller needs to configure stages along the flow's paths. We start by creating just one flow to measure its latency. Then we ask the controller to create an increasing number of flows to measure throughput. Intuitively, we expect throughput to benefit from batching several flow creation operations into one operation to the stages. Each flow's policy is dynamic point-to-point, like P2. One hypervisor and one storage server machine are used, with one SMBc and one SMBs stage in each respectively.

Creating a flow involves the controller reading the flow policy from a file on disk, then creating a new queue at each stage, mapping IO headers to that queue and setting the queue properties. Figure 11 presents the results. Creating a single flow takes on average 88 ms (dominated by the time to read the policy file). Batching flow creations is beneficial until a batch size of 1000 flows, beyond which batching does not help further. The peak throughput observed (not shown in the graph) is 13000 flows created per second, using a batch size of 2500 flows. At that point, the CPUs of the storage server stage SMBs are saturated. The controller itself is not a bottleneck and only 0.3% of its CPUs are utilized.

Figure 12 shows soft-state memory and network overheads at the controller and at one SMBs stage. As seen from the Figure, the memory overhead is low (at most 17 MB at the controller and 10 MB for the combined driver and slave at the server). The network overhead includes updates from controller (creating queue rules and configuring queue properties) and statistics the controller collects from stages every control interval. When creating 10000 flows the network overhead is 2 MB. When querying them for statistics the network overhead is around 0.4 MB/s.

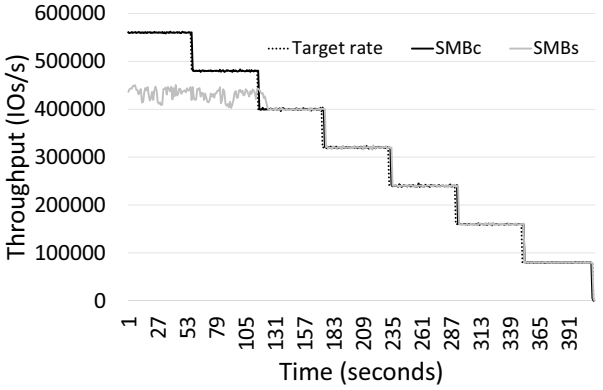

Figure 13: Convergence time for achieving target rate when enforcing at SMBc or SMBs stages. Best place to enforce is SMBc stage. Note that y-axis is in IOs/sec and each IO is 0.5 KB in size.

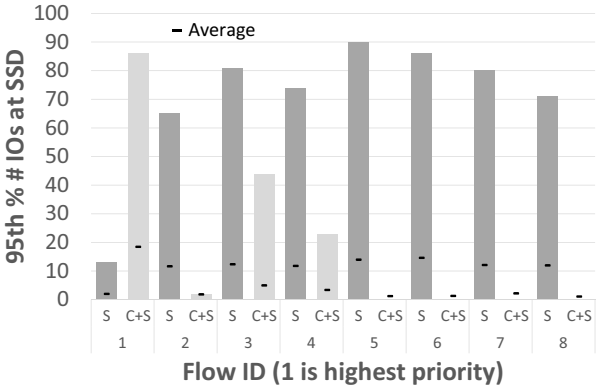

Figure 14: Graph shows 95^{th} percentile and average number of outstanding IO requests on the SSD backend. Flow ID 1 is the high priority flow. *S* refers to enforcement on the storage SMBs stage whereas *C* refers to enforcement on the hypervisor SMBc stage.

Convergence time. We measure next the lag between the time the controller changes a drain rate at the SMBc stage and the time the target rate is achieved end-to-end. To do so we use the 120 VMs across 10 hypervisors setup and the RAM store. The controller changes the policy every minute to reduce the previously allocated bandwidth by 81920 IOs/s. Figure 13 shows the achieved rate in IOs/s. All rates converge to the target ones in less than the control interval of 1 second.

6.2.3 Deciding where to enforce

The controller has global visibility and can choose where to enforce a policy. One heuristic it uses is to distribute the enforcement to minimize its performance overhead. For example, enforcing a multi-point bandwidth policy like P5 can be done either at the storage server SMBs stage, or across all hypervisors' SMBc stages. Since the storage server is a single enforcement point, enforcing there can lead to non-negligible overheads at very high rates. To measure this, we repeat the convergence time experiment, this time enforcing at the SMBs stage. Figure 13 shows the achieved rate in IOs/s. At high IO rates, we observe a 20% drop between the average target and achieved rates. The storage server's CPU is already saturated and adding the rate control processing leads to this drop. Using the above heuristic the controller is able to enforce at the SMBc stages and avoid the performance overhead.

The controller may choose distributed enforcement for reasons other than efficiency. Some policies, like P4 that offers high priority, are not amenable to single-stage enforcement. Figure 14 shows that this policy is best enforced at two stages rather than one. The setup is as follows: 8 identical VMs generate IOs from a single hypervisor to the SSD store. Flow ID 1 is assigned a high priority. The other flows have the same low priority. We expect a high priority flow to have the highest number of requests outstanding in the SSD queue.

Each IoMeter uses 8 KB random-access reads and issues 128 requests at a time (the 8 SSDs can process requests in parallel hence the plotted numbers are usually less than 128) over a period of 60 seconds. Each time an IO arrives in the system, the number of outstanding IOs at the SSD is logged and the 95^{th} percentile is calculated at the end of the 60 seconds.

The figure shows that enforcing the policy at the storage server only can lead to head-of-line blocking at the hypervisor, thus effectively negating the priority of the flow. The controller enforces the policy by having both stages apply priority treatment.[5]

7 Related work

The closest related work is on centralized control algorithms and APIs in software-defined networking (SDN) designs that decouple the data and control plane for network devices (NICs, switches, routers) [5, 6, 12, 21, 24, 32]. SDN designs build upon data plane primitives that have long been taken for granted in networks, such as traffic classification based on source and destination addresses, forwarding tables and queues. OpenFlow [13] extends and offers a standard interface to these primitives. Analogously, we have built a similar set of programmable data plane primitives and a logically centralized control plane for the storage IO stack. This required addressing challenges pertaining to storage.

[5]Note that in this simple scenario with one hypervisor, enforcing priority at SMBc would suffice (not shown in the figure), but that does not hold true when there are multiple hypervisor machines.

This paper also proposes a simple graph-based API exposed by the controller that we have used to build control applications. This API could benefit from policy languages, abstractions and compilers on top of current SDN stacks [6, 17], with benefits such as formal reasoning and automatic policy conflict checking.

SEDA showed how an *application* can be built as a series of stages with queues and controllers [30]. The focus there was on event and thread-driven ways to build high performing systems. Similarly, Click [11] allows software routers to be composed from modular elements. We have a very different focus in enabling control for the IO stack and the scope is a data center setting. Nonetheless these efforts resonate with our flexible queues and control building blocks.

A key challenge we address in this paper is mapping high-level IO identifiers to stage-specific IO identifiers to classify traffic end-to-end. Recent related work by Mesnier et al. [14] has applications label IO requests. The label then propagates with the request from the file system to a block device using the SCSI protocol. Others have also used explicit labels for IO requests in distributed systems [23,27]. IOFlow does not require applications to label their IO or any system protocol changes to support the extra label. However, it can make use of such labels if they already exist and set per-stage queuing rules based on them.

Tenant performance isolation is a key controller application in this paper that illustrates the benefits of the architecture. The isolation problem itself has been extensively studied and there are many customized solutions in the context of several resources: for example, for disks [9, 28], for multi-tenant network control [2, 3, 20], for multi-tenant storage control [8, 25], for latency control in networks [1] and for multi-resource centralized systems [7]. Distributed rate limiting has also been studied in the past [22]. In contrast to these proposals, IOFlow offers a single framework for a wide range of performance isolation policies. The presence of a controller with global visibility and the programmable dataplane stages allowed us to write simple centralized algorithms to achieve the policies. As such, we believe that other algorithms, such as recent ones by Shue et al. [25] would equally benefit from our system's support.

8 Conclusion

This paper presents IOFlow, an architecture that enables end-to-end policies. It does so by decoupling the control of IO flows from the data plane and by introducing programmable data plane queues that allow for flexible service and routing properties. IOFlow extends SDN designs, and allows IO control close to source and destination endpoints: the hypervisor and storage server in this instance. This control allows for high-level policies expressed in terms of a four-tuple: $\{$*VMs, operations, files, shares*$\}$. A key strength of the architecture is that it does not require application or VM modifications. Through two control applications, we show that IOFlow enables useful policies that are hard to achieve today.

9 Acknowledgments

We thank the anonymous SOSP reviewers, our shepherd Robbert van Renesse, and others, including Dushyanth Narayanan, Jake Oshins and Jim Pinkerton for their feedback. We thank Andy Slowey for helping us maintain the computer cluster. We thank Bruce Worthington and Swaroop Kavalanekar for the Hotmail IO traces.

References

[1] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of Usenix NSDI*, San Jose, CA, 2012.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM*, pages 242–253, Toronto, Ontario, Canada, 2011.

[3] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawaradena, and G. O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *Proceedings of Usenix NSDI*, Lombard, IL, 2013.

[4] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1992.

[5] 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*, pages 1–12, Kyoto, Japan, 2007.

[6] 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.

[7] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM*, pages 1–12, Helsinki, 2012.

[8] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proccedings of Usenix FAST*, pages 85–98, San Francisco, California, 2009.

[9] A. Gulati, A. Merchant, and P. J. Varman. mClock: handling throughput variability for hypervisor IO scheduling. In *Proceedings of USENIX OSDI*, Vancouver, BC, Canada, 2010.

[10] Intel Corporation. Iometer benchmark, 2013. `http://www.iometer.org/`.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[12] 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.

[13] 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.

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

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

[16] Microsoft Corporation. Scanner file system minifilter driver (MSDN), 2013. `http://code.msdn.microsoft.com/`.

[17] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *Proceedings of Usenix NSDI*, Lombard, IL, 2013.

[18] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of Usenix NSDI*, Lombard, IL, 2013.

[19] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of ACM SOSP*, pages 29–41, Cascais, Portugal, 2011.

[20] L. Popa, A. Krishnamurthy, G. Kumar, S. Ratnasamy, M. Chowdhury, and I. Stoica. FairCloud: sharing the network in cloud computing. In *Proceedings of ACM SIGCOMM*, Helsinki, 2012.

[21] 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.

[22] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, 2007.

[23] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of Usenix NSDI*, Boston, MA, 2011.

[24] SDN Central Community. SDN Central. `http://www.sdncentral.com/`.

[25] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of Usenix OSDI*, pages 349–362, Hollywood, CA, USA, 2012.

[26] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys*, pages 169–182, Salzburg, Austria, 2011.

[27] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of ACM SIGMETRICS*, pages 3–14, 2006.

[28] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings Usenix FAST*, San Jose, CA, 2007.

[29] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of ACM SoCC*, San Jose, California, 2012.

[30] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of ACM SOSP*, pages 230–243, Banff, Alberta, Canada, 2001.

[31] Wikipedia Encyclopedia. Token bucket, 2013. `http://en.wikipedia.org/`.

[32] 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.