Blender: Upgrading Tenant-based Data Center Networking

Kevin C. Webb
kwebb@cs.swarthmore.edu
Swarthmore College

Arjun Roy, Kenneth Yocum, and Alex C. Snoeren
{arroy, kyocum, snoeren}@cs.ucsd.edu
University of California, San Diego

ABSTRACT

This paper presents Blender, a framework that enables network operators to improve tenant performance by tailoring the network's behavior to tenant needs. Tenants may upgrade their provisioned portion of the network with specific features, such as multi-path routing, isolation, and failure recovery, without modifying hosted application code. Network operators may differentiate themselves based on upgrades they offer, creating new upgrades via a lightweight programming interface. Blender safely executes multiple tenants' selections simultaneously across a shared network infrastructure. We show that the Blender model can express and extend recently proposed network functionality on existing SDN networks. We use an OpenFlow-based prototype to quantify Blender's performance and potential for deployment at scale.

Categories and Subject Descriptors

C.2.1 [**Network Architecture**]: Network topology; C.2.3 [**Network Operations**]: Network management

Keywords

network isolation; multiple tenants; SDN

1. INTRODUCTION

Data centers are rapidly evolving to accommodate the performance demands of the cloud computing model, in which raw computing resources are provisioned for users on demand. However, despite the recent explosion in the popularity of cloud computing services, the underlying data center network remains difficult to virtualize. While all users expect the network to provide basic packet forwarding, they have contradictory preferences regarding the behavior of supplemental functionality like performance isolation, latency management, multi-path route selection, failure recovery, and so on. Despite many strong proposals to augment existing networks with such features, each tackles one part of the problem in a disparate way, with no prevailing unifying approach.

This work presents Blender, a framework that supports tenants mixing desired network functionality. Regardless of whether a data center is public, i.e., its resources are rented out to anyone willing to pay, or private, supporting only the services of the data center's owner, we make a distinction between the infrastructure provider and their tenants. Each actor holds a vested interest, from a different perspective, in the network provisioning process. The *tenants*, who use their allotted resources to support Web, corporate, social media, or other user-facing services, desire a mechanism to tailor the network's behavior to their specific needs, ideally without modifying their applications. The data center *providers*, who own and manage data center resources, wish to cater to tenants' requirements to differentiate themselves from competition, attract new customers, and increase revenue. However, since network design and ensuring tenant privacy provide competitive advantages, providers often prefer to limit tenant visibility into their networks, opting instead to supply tenants with a choice of pre-approved components. In Blender, providers can create network *upgrades* that each export functionality and network visibility in accordance with their business model.

On one hand, operators wish to support a wide variety of tenants with different performance and reliability demands [7]. On the other hand, they want to reason about network bandwidth and latency in a manner consistent with the isolated, chargeable units of CPU, disk, and memory that server virtualization provides. A consequence is that cloud providers today, such as Amazon, only make qualitative (e.g., low, moderate, high) assurances for network performance. With unpredictable network loads, today's data center operators must accept hot spots (i.e., tenants pay to wait [35]) or monitor and dynamically adjust VM placement [20], application components [39], or network flows [2] to improve utilization.

Using Blender, tenants augment their network environment by selecting and applying only the upgrades that best meet their needs. As an example, consider three tenants that share the physical data center network of a public cloud computing platform: a tenant managing a three-tiered Web service, a tenant executing a bulk data processing job (e.g., MapReduce or Hadoop), and a tenant hosting back-end business logic. Each tenant might prefer a different form of performance isolation, ranging from fully opportunistic work conservation [33] for the bulk processing tenant to reserved, predictable performance [5] for the business logic tenant. Furthermore, each tenant would benefit from a different set of supplemental upgrades to support their execution: the Web service may require bounded latency [38] to meet customer performance objectives, while the bulk processing service may fare better with support for performance-aware flow placement [2]. Such flexibility allows operators to better support tenant needs while differentiating their cloud offerings through custom resource management and charging models.

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.

ACM 978-1-4503-2839-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2658260.2658268.

In designing such a system, we face two key challenges. First, we must ensure that the programming model used by network providers to build upgrades balances simplicity with the ability to express complex functionality. To do so, Blender defines a concise set of abstractions that capture a range of upgrades, while also enabling the run time to check and ensure that the underlying network can support the co-existence of multiple upgrades. The second primary challenge is scale. Blender must account for the finite resources available in switching hardware for managing traffic and storing forwarding entries. We overcome these hardware capacity limitations by combining a specialized tenant model with optimization techniques to prevent resource exhaustion.

This paper makes the following contributions:

• **Blender framework:** Blender supports multiple network tenants, each of whom may simultaneously deploy many upgrades on a shared physical network. Network providers use a programming model that exposes eight high-level network attributes, such as routes and rate limits, to create upgrades. Ultimately upgrades compile into a set of resource reservations across the physical network. Many upgrades use few attributes, maybe 3-4, and this is sufficient to provide upgrade modules that provide fixed or proportional network isolation. In addition, upgrades may work in concert with one another, providing work conservation, deadline-aware flow scheduling, dynamic flow placement, and other services.

• **Blender network architecture:** Blender provides this flexibility while ensuring a consistent and scalable forwarding infrastructure. It multiplexes network resources at the granularity of network tenants, which can represent an entire service or distinct applications with specific network demands. Transactional tenant allocation ensures atomic and isolated changes to network forwarding state.

• **Implementation and evaluation:** We illustrate these concepts through an OpenFlow-based [19] prototype. We demonstrate the ability to author, compose, and execute multiple network upgrades, including functionality found in recent work [2, 5, 26, 33, 38]. Blender leverages switch-based traffic policers to simplify resource allocation and dynamic traffic control; we describe the resource requirements for tenant allocations and show that for realistic resource allocation strategies, per-switch rule and policer counts are bounded by $O(\text{numTenants} * \text{portCount})$. Even without those assumptions, our tenant-churn experiments on a 50-node, 200-VM, fat-tree testbed easily fit within the resource constraints of HP's prototype OpenFlow switches.

2. RELATED WORK

Multi-tenant SDNs. Blender is one of several systems that provision shared network infrastructure in support of multiple tenants and applications. FlowVisor [32] provides strict tenant separation by dividing the network into independent "slices", each of which maintains management routines in the form of a private SDN controller. Each tenant is free to choose, and is responsible for providing and managing, custom SDN software within their assigned slice. At the other end of the spectrum, Onix [17] provides an all-purpose controller framework designed to be shared by simultaneously executing applications. Akin to Blender, Onix provides a graph-based model of the physical network for applications to manipulate, but it requires that tenant applications mediate their own interactions.

CloudNaaS [6] allows tenants to select their desired functionality similarly to Blender, though it emphasizes end-host naming and addressing, accepting verbose network specifications in a rich language. In contrast, Blender focuses on developing a model for combining network features based on short, high-level tenant requests. In Participatory Networking (PANE) [10], a centralized controller provides an API with which modified tenant applications can obtain network visibility and reserve link capacity. Like Blender, PANE hierarchically subdivides network resources and resolves conflicting requests with resource-specific routines. PANE exposes network information to tenant applications and requires them to engage in custom resource provisioning. With Blender, tenants make brief requests prior to executing unmodified applications in a fashion similar to a tenant's virtual machine requests from PaaS cloud providers.

Network performance isolation. Multi-tenant data centers balance the need to run the network at maximum efficiency with the desire to provide performance guarantees for individual tenants. Some schemes provide predictability via fixed performance guarantees [5, 14], but limit the number of concurrent tenants placed on the network. Others maintain proportional network shares, allowing tenants to receive performance relative to tenant demand [18, 28, 33]. These are fundamental trade-offs that any isolation model must make—no single network allocation strategy can provide every isolation property desired by a tenant or network operator [24].

Blender resolves this situation by allowing network operators to create and deploy multiple isolation models across a shared infrastructure. As opposed to monolithic approaches, tenants are free to choose the model that best represents their needs. We demonstrate this flexibility in our prototype by implementing two performance isolation upgrades, inspired by Oktopus [5] and Seawall [33] using Blender's upgrade programming model in Section 4.

Augmenting network functionality. In addition to performance isolation, Blender allows operators to offer supplemental feature upgrades to tenants. Recent proposals have demonstrated the benefits of supplying cloud tenants with functionality like latency control [3, 38], flow placement [2], load balancing [4, 36], middleboxes [25, 30, 31], and other services. Blender allows tenants to request such services via a unified resource request model, and we describe several examples upgrades based on D3 [38], Hedera [2], and DRL [26].

Network programming. Blender is far from the first system to empower applications with explicit control over the behavior of the network fabric. For example, active networking [37] provides differentiated network behavior in response to user-supplied forwarding directives. In Blender, however, tenants make concise, high-level resource requests—as opposed to active networks' per-packet model. Contemporary to active networks, Tempest [29] allows users to create virtual private networks over an ATM substrate. Tempest partitions switches into "switchlets", on which users may execute forwarding programs.

More recent projects address the need for higher-level languages for constructing and maintaining SDN controller software. Nettle [34] is a domain-specific, functional language embedded in Haskell that allows network operators to write declarative programs for reacting to OpenFlow network events. Frenetic [11] combines a declarative query language for classifying and aggregating network traffic with a functional library for describing reactive packet-forwarding policies. Like Blender, Frenetic enables the network to compose reusable software modules, however it does not provide explicit support for multi-tenancy or performance isolation. Pyretic [21] extends the ideas of Frenetic with an imperative Python-like language for sequentially composing modules that can process packets with virtual traffic headers. These efforts enable the composition of applications from the parallel and sequential execution of modular components, but efficiently compiling their directives into forwarding hardware remains challenging.

Table 1: Blender's programming attributes.

Attribute name	Conflicts	Allows upgrade to ...
Create zones	Static	Sub-divide network regions.
Choose routes	Static	Establish VM reachability.
Create rate limits	Dynamic	Add or remove rate limits.
Modify rate limits	Dynamic	Change the enforced rate limit.
Read statistics	N/A (r/o)	Receive network traffic information.
Assign flow paths	Static	Bind a flow to a set of hops.
Intercept traffic	Static	Inspect packet content.
Modify traffic	Static	Re-write packet content.

Table 2: Upgrade allocators manipulate graphs G that contain the following logical elements. Here links and switches are given unique identifiers: l_{id} and s_{id}.

Name	Item Definition
Graphs	G :={Switches,Links}
Links	l :={l_{id},`capacity, weight, state dict`}
Switches	s :={s_{id}, `state dict`}
Assignments	res :={AsnType,l_{id},Traffic,args}
AsnType	rt :={`<upgradeID>`}
Traffic	tc :={pattern}

3. THE BLENDER FRAMEWORK

Blender allows network providers to specify network functionality using a small number of high-level abstractions that we call *upgrades*. A logically centralized controller executes upgrades at the request of tenants and ultimately translates their directives into forwarding and rate limiting state within a configurable subset of the network, allowing multiple upgrades to compose and execute across the same physical infrastructure. We require that the cloud provider—rather than the tenants—author and supply upgrades to ensure that they are not malicious or otherwise abusive to the network.

From the provider perspective, Blender's programming model abstracts the features and functionality of the network forwarding hardware, for example, rate limiting, providing traffic statistics, selecting flow paths, modifying packets, etc. Despite many devices implementing and exposing these features differently (e.g., in switch hardware vs. at end-host hypervisors), Blender exports a single interface for each resource; the runtime translates these calls for the particular underlying device(s) in a given network. Blender requires the network provider to enumerate the set of available features and their interface definitions. While providers are free to define their own attributes to match their hardware, environment, and goals, Table 1 summarizes the abstract attributes in Blender's programming model (API).

When providers develop an upgrade, they must label the upgrade with the required attributes to inform Blender of how the upgrade will be using network resources. Labeling an upgrade as 'using an attribute' provides two primary benefits. First, it provides the upgrade with access to the attribute's programming interface, expanding the set of events or network state updates for the upgrade to utilize (Section 3.2). Additionally, labeling assists in conflict detection (Section 3.3).

To tenants, the implementation details of upgrades are hidden. Instead, a tenant specifies a list of desired upgrades by submitting a small, high-level request to Blender's centralized controller. The request includes the number of hosts in the virtual network and the list of upgrades the tenant wishes to use (along with their parameters). Blender applies the upgrades without intervention from the tenant's application. In general, tenant applications need not be modified, except in the case of upgrades that explicitly interact with tenant code as a part of their design (e.g., deadline-aware scheduling, Section 4.3). Note that a default virtual network (one with no upgrades) provides best-effort connectivity over a spanning tree.

3.1 Tenant allocation

Before a tenant can begin using the network, Blender must initialize each of the tenant's selected upgrades. The instantiation of each upgrade begins with a static allocation phase followed by an optional runtime component. Allocation determines the structure of, reserves capacity for, and instantiates a tenant's virtual network. If the upgrade must react to tenant's traffic in real time, Blender allows it to subscribe to event notifications exposed by specific network attributes in the programming model (Table 1).

Thus for a new tenant request, each upgrade analyzes how to meet the tenant's objectives and generates a proposed set of network changes. Blender then determines whether the changes are allowed and feasible, and if so, admits the tenant, configuring the underlying network to enforce the directives. Note that this might result in reconfiguration of the rate limits for other tenants' virtual networks, e.g., to enforce proportional bandwidth shares[1].

Blender represents resources as a logical *graph* of switches/hosts (vertices) and links (edges). Blender and its upgrades use these graphs to maintain state on behalf of the tenant's virtual network. The top half of Table 2 describes the graph, link, and traffic assignment elements to which upgrades have access. Links and switches contain a `state` annotation, a provider-defined dictionary that allows upgrades to maintain bookkeeping information across tenant arrivals and departures.

Capacity reservations. While all upgrades go through static allocation, reserving link capacity will typically be the responsibility of only one *isolation* upgrade, selected from a set of mutually exclusive upgrades that provide inter-tenant performance isolation. When a tenant submits a request, its isolation upgrade computes paths to connect the tenant's endpoints and creates capacity reservations via the API's rate limiting attribute to describe how to treat the tenant's flows.

The API's rate limiting attribute supports fixed-rate and proportionally weighted bandwidth reservations. By default, such reservations are not work conserving, however work conservation can easily be introduced by the inclusion of an additional upgrade.

The tenant's selected isolation upgrade claims resources by modifying the state of its input graph according to the type, link, and any additional arguments the reservation may require (e.g., capacity or weight). For example, for an upgrade to create a proportional reservation with a weight of w and associate it with link l the rate limiting attribute increments l's weight in the input graph by w and later installs policers to rate limit traffic over l to $l.\texttt{capacity} * \frac{w}{l.\texttt{weight}+w}$.

The isolation upgrade takes a tenant t and a network graph G as input. The upgrade is restricted to using only the resources described in G, even if additional resources may be physically available[2]. The upgrade analyzes the graph and computes a new graph representing the tenant's desired resources: $allocate(G,t,\ldots) \rightarrow G'$. If Blender admits the tenant, it *removes* the resources in G' from G. Subsequent allocations may either use the remaining re-

[1]This work considers Blender in the context of a trusted environment; providers install upgrades that do not sabotage other tenants.
[2]This mechanism may be combined with ACLs or other high-level resource management policies to control resource access.

Table 3: The Blender upgrade API for managing event subscriptions and performing network state updates.

Function	Description
subscribe(event, location, cb func)	Registers for event callbacks.
unsubscribe(event, location)	Unsubscribes from a callback.
update(attribute, location, [args])	Updates attribute network state.

sources in $G \backslash G'$ or subdivide those allocated to G'. This ability to pass modified graphs through successive allocations supports online allocation of sub-tenants. Thus, the sequence of tenant arrivals and departures forms a logical hierarchy of network graphs.

When hierarchically composing capacity reservations, we impose a simple restriction: We allow fixed allocations to sub-divide a fixed allocation, and proportional allocations may also sub-divide a fixed allocation, but never the opposite. Thus, a proportional allocation can be used to sub-divide a fixed allocation, but any future allocation after the transition to proportional must also be proportional.

Assigning upgrades to traffic. When a tenant's virtual network has been provisioned by the isolation upgrade, the remaining upgrades apply their functionality to tenant traffic and subscribe to attribute events by annotating the tenant's network graph with traffic assignments as shown in the bottom half of Table 2. Note that each link on a given network graph may have multiple traffic assignments, one for each upgrade, and those assignments are not required to be exclusive. This supports combining a fixed capacity allocation with features like work conservation and traffic-aware flow assignment. Upgrades record their assignments by annotating the state dictionaries associated with the tenant's graph. The annotations are later used to compile the tenant into hardware directives (Section 5).

3.2 Runtime upgrade execution

After allocation, the tenant may start using the virtual network, and upgrades may begin their runtime execution. While some upgrades use only static API attributes (e.g., those that reserve only fixed capacity), many upgrades wish to adapt to dynamic network conditions. Blender supports such functionality by allowing upgrades to subscribe to network events (e.g., link failures or new flows starting) and to define a control loop that continues to execute for the lifetime of a tenant. Table 3 describes the API available to upgrades for subscribing to events and updating network state.

Unlike the static allocation stage, upgrade control loops do not manipulate tenant network graphs. Instead, they interact directly with the underlying network equipment to monitor and modify the state of the network. Upgrades receive notifications when their events are triggered. An upgrade may execute an arbitrary callback function in response to an event notification, where the upgrade may choose to change its event subscriptions or update network state. We describe example upgrades that take advantage of dynamic events in Section 4.

3.3 Network attribute conflicts

In Blender, a conflict occurs when two or more upgrades attempt to modify the network, using the same network attribute, in a manner that is unsafe without external coordination. To account for the differences in attribute behavior, the specific conditions that represent a conflict are attribute-dependant. Blender uses spatial *zone* separation to reduce the likelihood of conflicts and performs attribute-specific resolution for conflicts that cannot be avoided. Zones are defined hierarchically, as subsets of the graph representing the tenant's network devices and links. Blender expects tenants to specify in which zone each of their upgrades will execute. We provide a special zone creation attribute that allows for the definition of new zones. By convention, our implementation restricts this attribute to isolation upgrades.

Even with zones scoping the use of attributes, multiple upgrades may wish to share a particular attribute within a single zone. When such a conflict is discovered, Blender executes a provider-defined, attribute-specific resolution routine[3]. The conflict resolution routine is free to take any information about the state of the network or the tenant's upgrade set into account. Depending on the cloud provider's policies and the attribute in question, resolution strategies may involve prioritizing one upgrade over another, processing each upgrade sequentially, coordinating the upgrades such that they safely share network resources, or rejecting the request. For upgrades that require coordination, the requirement is enforced in the attribute API they use.

The Blender model makes a distinction between two forms of conflicts, static and runtime conflicts, which correspond to the two stages of upgrade execution. To ensure safe execution, a static conflict is one that must be detected and resolved during the upgrade allocation stage. Consider an example in which a tenant requests multiple upgrades that assign flows to routes. We leverage attribute labeling to detect this condition, and for this example, resolve the conflict by rejecting this (nonsensical) tenant request. With a different attribute, for example modifying traffic, Blender may choose to resolve a conflict by ensuring a sequential ordering on the flow of traffic through the set of conflicting upgrades.

For some attributes, the presence of a conflict may depend not only on upgrades sharing a zone, but how the network is managed by the upgrades within that zone. For example, if multiple co-located upgrades declare that they modify rate limits, whether or not they conflict depends on their traffic assignments. For network attributes whose API have additional parameters, like traffic assignment, the network provider may opt to resolve their conflicts at runtime. These runtime conflicts are detected and resolved by the network provider's implementation of attribute `update` calls.

4. SAMPLE UPGRADES

This section demonstrates Blender's flexibility in expressing a variety of upgrades. We take inspiration from recent proposals and create two network performance isolation upgrades along with upgrades for flow path assignment, work conservation, deadline-aware scheduling, failure recovery, and distributed rate limiting.

4.1 Performance isolation upgrades

Our prototype includes a pair of performance isolation upgrades: Squid and Jetty. Squid creates fixed-capacity, predictable allocations in the spirit of Oktopus [5] and FairCloud's PS-P [24] models, allocating over-subscribed virtual network topologies. Jetty creates proportional allocations in the fashion of Seawall [33] and PS-L [24], providing isolation at the link level. Our descriptions of these upgrades focuses on our adaptation of the models and their interaction with the Blender framework. We refer the reader to the original proposals of these models for additional details of their operation.

Squid: predictable virtual networks. Squid mimics the data center isolation model provided by Oktopus [5] and PS-P [24],

[3]Blender's per-attribute conflict resolution routines are similar to those of PANE [10], which handles conflicts on a per-'atom' basis.

which allows tenants to request virtual networks that emulate non-blocking switches. Each switch i connects n_i VMs with bisection bandwidth b_i. The switch may be further connected to other virtual switches, forming clusters, based on an over-subscription factor O. Thus, a virtual switch must have uplink capacity $(b_i \cdot n_i)/O$ to each other switch. Like Oktopus, we assume the input graph is a singly-rooted, multi-level tree. A complete Squid request is specified as $(G, \texttt{squid}, C_{VMs}, j, n, b, O)$.

This request will cause the Squid upgrade to embed j virtual switches, each connecting n VMs with links of capacity b, in the input network graph G. Squid will arrange the virtual switches to have an over-subscription ratio of O. The upgrade is free to choose the $n \cdot j$ endpoints from the provided $C_{VMs} \in G$ candidates.

Squid uses a first-fit strategy to find feasible virtual switch (cluster) placements. It uses a depth-first recursive algorithm that attempts to place each cluster as low in the topology as possible, minimizing the number of links traffic crosses. For each switch in the topology, Squid records the number of clusters that can be placed beneath it. To determine if a given cluster of size n_i can be placed under a switch, Squid first checks the links l in G for sufficient capacity b to connect a cluster. It then ensures that each found cluster has capacity $\frac{b_i * n_i}{O}$ to every other feasible cluster. If sufficient capacity does not exist (or if too few candidate endpoints were supplied), the Squid upgrade rejects the tenant.

If successful, this process returns a subtree under which all clusters fit, and each cluster is considered a separate zone. Squid then traverses this subgraph from the top down, creating a fixed bandwidth reservation and assigning additional upgrades to each link. The capacity it reserves is the sum of the connectivity requirements of all clusters under this link.

Jetty: proportional tenant allocations. Jetty provides proportional link capacity sharing similar to PS-L [24] at the granularity of tenants (in contrast to Seawall's entity-based proportional shares [33]). For each tenant, Jetty ensures a proportional share of capacity on each physical network link the entity uses. While we simplify our discussion of Jetty by assuming a singly-rooted, multi-level tree topology, it may be trivially extended to multi-rooted topologies. A Jetty request consists of $(G, \texttt{jetty}, C_{VMs}, n, w)$.

Jetty begins by finding the n endpoints in C_{VMs} that are best connected to the network core. It calculates the bottleneck bandwidth on the shortest path from the top-level core switch to each endpoint and then chooses the top-n highest-capacity endpoints. Unlike Squid, Jetty never rejects requests (assuming $C_{VMs} \in G$ contains at least n endpoints). Like Squid, Jetty notes the lowest level root switch under which all VMs connect, and it creates reservations in a similar top-down process. Each reservation is proportional and uses the same weight w.

4.2 Performance isolation discussion

It is instructive to discuss how Squid and Jetty differ from their inspirations, which install traffic policers only on sending VMs. Our upgrades use in-network traffic policers in addition to end-host rate limiting. This decision allows Squid and Jetty to collapse the collection of tenant traffic across each reservation and eliminates the need for the tightly coupled VM sending rate coordination found in Oktopus. Under our scheme, the upgrades create at most one reservation per link per tenant, meaning a 48-port switch connecting hosts with 8 VMs would require at most $2*8*48 = 768$ rate policers (assuming the worst case scenario in which every VM is in a different tenant). This is well within the hardware policer count of modern top-of-rack switches; for example Cisco's 4948 E/F top-of-rack switch supports as many as 8K policers [9]. Rule optimization (Section 6) often also allows Blender to combine the policers for multiple tenants, further reducing the hardware requirement.

4.3 Functionality-enhancing upgrades

We leverage Blender's modularity to design a variety of upgrades that improve the performance and capabilities of tenant virtual networks. At the tenant's request, Blender mixes these upgrades with the reservations applied by the isolation upgrade. For instance, a work-conserving version of Squid simply pairs a work conservation upgrade with each fixed reservation.

Flow path selection. When multiple paths connect a pair of communicating endpoints, Blender selects only one route for any individual flow that passes between them to avoid TCP reordering problems. Here, we describe two upgrades we have built for assigning flows to paths. The first upgrade, random assignment (RA), randomly maps each flow to an available route, which closely approximates conventional ECMP [16]. The second, bandwidth-maximizing assignment (BMA), is inspired by the global first-fit algorithm of Hedera [2]. BMA assigns a flow to the route with the largest available capacity.

Both flow assignment upgrades are similarly structured, and we label each with the 'assign flow paths' attribute. This label enables upgrade subscription to a 'new flow' event notification, which is triggered by the end host hypervisor when a flow begins at one of its VMs. RA simply chooses a random path in response, whereas BMA additionally utilizes the 'read statistics' attribute to read and compare path utilizations before returning a decision. While they currently perform placement for all new flows, either could easily be extended to selectively optimize only long-running or high-volume flows for bandwidth maximization.

Work conservation. Work conservation allows a tenant to contribute the unused portion of a bandwidth reservation on a link to an excess capacity pool. If other tenants on the same link are operating below their reserved capacities, capacity-hungry tenants can draw capacity from this pool. Note that the work conservation upgrade described here is implemented by querying for traffic demands and modifying the rate limits of the hardware policers that are already associated with the tenants' reservations. It does not require weighted fair queueing or any other explicitly work-conserving hardware mechanisms. This design also illustrates the flexibility of upgrades, as the excess capacity pool may be shared among all reservations on the link, regardless of their upgrade, or it may be scoped to a subset of the link's reservations.

Deadline-aware flow scheduling. While the conventional performance metric for data center applications has been bandwidth, several recent projects [3, 38] have begun to focus on latency. This work is largely motivated by reports from industry [15], which indicate that even small increases in latency can lead to a significant reduction in customer satisfaction.

For the Blender prototype, we constructed a deadline-aware upgrade that was inspired by D3 [38], which uses deadlines to schedule flow completions. Similar to D3, making use of the upgrade requires explicit interaction from the tenant application, possibly requiring application modifications. The tenant application informs its local shim of the size and deadline for each new flow that it starts. The shim then periodically computes the rate necessary for the flow to complete on time and issues a request for that capacity to the upgrade. At the same interval, the upgrade reviews its requests and produces a schedule of flow rates, which it uses to update switch rate policers.

Like D3, our flow scheduling upgrade would ideally execute directly within the network's devices. Unfortunately, our experimental switch platform was not optimized for fast rate policer updates and could not keep up with the upgrade's proposed changes. The poor performance is an issue of software (fast rate policer updating is not a frequently requested feature) and not a fundamental limitation of the hardware. To stand-in for programmable switches, we substitute the controller to execute the upgrade, which reduces the responsiveness.

Failure recovery. As the size of a network increases, so does the likelihood of a component failure. We recognize that different tenants may wish to respond to such failures in numerous ways, and the Blender model allows for such diversity. For example, one tenant may want to minimize the service interruption by automatically routing around the failure, a second tenant might want to allocate a new virtual network in a different physical location, and a third may wish to halt execution until the failure is resolved.

For our Blender prototype, we construct a failure recovery upgrade that executes at the controller and operates like the first of the three examples described above. When a link failure occurs, the upgrade determines which of a tenant's routes crossed that link and searches for an alternative path, among the network's free resources, whose annotated characteristics indicate that it could serve as an equivalent replacement. Upon finding such a link, it then applies the reservations from the failed link to the replacement and informs the appropriate end hosts to mark their future packets to use the replacement (Section 6.2).

Distributed rate limiting. DRL [26] provides a mechanism whereby multiple senders coordinate their sending rates to ensure that their aggregate traffic rate is below a maximum global rate limit, without statically limiting their rates. The DRL upgrade periodically checks the rates, across multiple links, of a coordinated set of traffic and adjusts link-local rate limits in proportion to the traffic's demand across those links. The application of DRL is useful in a situation in which a tenant would like to control the aggregate sending rate across paths that may not share any common hops. For example, in a multi-tiered, distributed service, a tenant may want to limit the global rate at which one tier can transmit to another.

5. ARCHITECTURE

This section describes Blender's architecture that compiles, installs, and monitors upgrades for multiple tenants. This design builds on a software-defined network (SDN) substrate whose switches accept rules that dictate how packets move through the network [8, 19]. In addition to the standard rule primitive that matches header fields and dictates output ports, we utilize traffic rate limiting primitives.

Unlike other SDN architectures that allocate at the granularity of flows [2, 8], Blender allocates resources to tenants as a whole. Thus, Blender needs to ensure two important forms of concurrency control. First, upgrades must receive a consistent view of the input network graph. A tenant should not observe other tenants' allocations and should execute only if it can receive all its resources. Second, a tenant's rules must be fully installed before sending traffic. This provides *per-tenant* forwarding consistency; all packets of a given tenant obey one set of forwarding and rate limiting rules at a time.

Blender provides these semantics by processing a tenant request as a transaction. Each allocation involves creating reservations, generating switch rules, optimizing the rule count (Section 6.3), and installing rules into the network. We call the first three steps

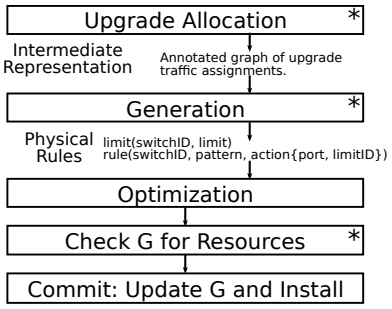

Figure 1: After allocation, compilation converts an intermediate representation of abstract limits and routes into a concrete set of rules. *'d stages can reject the tenant before it commits.

compilation as a tenant's routes and resource reservations must be converted into SDN-compatible forwarding rules. Figure 1 shows this sequence of events as Blender handles a tenant request.

Stages marked with an asterisk may abort the transaction before any changes are made to the physical network. For instance, SDN switches have limited storage capacity in their forwarding tables, and the Blender controller ensures that all network elements have sufficient space for the tenant's optimized rules. At this point Blender commits updated annotations to the tenant's input graph to reflect the tenant's resource claims. Blender commits tenants in the order it receives them. If two tenants attempt to allocate the same resources, one will ultimately commit before the other, causing the second to roll-back and either try again or be rejected, thus preventing tenants from executing without receiving their full set of resources.

Finally, after successfully committing updates to the available network resources, Blender pushes the tenant's rules into the network. To avoid the consistency issues described in [27], we wait until the tenant's network state is fully installed in all devices (i.e., consistent) before notifying the requesting tenant that the network is available and ready for use.

6. IMPLEMENTATION

This section describes an OpenFlow-based implementation of Blender. In the OpenFlow model [19], switches are simple forwarding elements whose forwarding tables are populated by an intelligent, logically-centralized controller. While we have not modified our OpenFlow switches, we are leveraging HP Labs extensions to our HP ProCurve switches. These allow an OpenFlow controller to define limiters on a switch and add a rate limiting `action` to OpenFlow rules that reference those limiters. This provides the necessary statistics and fine-grained control over switch packet forwarding and rate limiting.

6.1 Blender controller and rule installation

We implement the Blender controller in Python as a component for NOX [13], an open-source OpenFlow controller. It maintains Blender's state, including the physical network representation, allocated network graphs, connections to end-hosts, and the limiters and rules installed in the network devices. The remaining components of the system are implemented as modules within the controller, including the library of operator-installed upgrades. Useful upgrades may be written in a few hundred lines of code.

After Blender compiles and commits a tenant's virtual network, it installs routing rules and limiters on the switches. For performance, it must ensure that all rules reside in fast-path hardware

lookup tables. Our switches can only install a limited number of rules into their TCAM in a short period of time, and additional rules are silently added to a slower software table. To prevent these slow software rules, we rate limit our rule installation and periodically check for and move any rules that are found to be in the software table. We have empirically determined that issuing eight rules per second works well for our switches.

6.2 End-system shim layer

Tenants interact with Blender by submitting resource requests. They do so by communicating with a local Blender shim that is running in user-space on each VM. Shims maintain a communication channel with the Blender controller. Upon connecting to the controller, the shim transmits a unique identifier on behalf of the host it represents. The controller considers this the endpoint identifier and associates it with the physical location of the VM. While this binds the VM to a physical host in our prototype, it would be straightforward to move the shim into a hypervisor or adopt recent proposals for data center address virtualization [22].

Tenants submit their requests to the local shim, which relays them to the Blender task controller. After the controller installs rules and limiters at the switches, the controller replies to each shim in the resulting virtual network. The reply contains a set of type of service (TOS) bits and an optional rate limit for each destination in the tenant's network. Upon receiving this information, the shim manipulates `iptables` to mark outgoing packets with TOS bits and `tc` to rate limit flows. For destinations with multiple paths, the shim installs TOS marking rules according to the directives of the active flow path selection upgrade (Section 4.3), where the default is random assignment. If a tenant is rejected during the allocation phase, the shim will receive a callback to indicate this failure.

6.3 Rule optimization

OpenFlow rules eventually reside in switch memory, a limited resource. Our testbed switches store rules in a TCAM, which allows wildcard matching via "don't care" entries. TCAM sizes are limited due to cost and power requirements, and Blender's rule optimization phase tries to conserve space in these switch tables by combining rules into a single wildcarded TCAM entry, when appropriate. This improves the scalability of the system by allowing more (or larger) tenant networks to be installed.

Blender's optimizer leverages two abstractions to combine forwarding rules: our tenant-based allocation strategy and a hierarchical network identifier space, as enabled by several recently proposed virtualization systems [12, 22, 23]. For each switch port in use by a tenant, it uses wildcarding to collapse all of the port's rules into a match on the longest-prefix destination and TOS field of the rule set. The optimizer is space-efficient and produces only one forwarding rule per switch port per tenant. Unfortunately, our switches do not respect OpenFlow rule priorities, making longest prefix matching, and the use of this optimizer in our testbed, impossible.

We implemented a second optimizer that is simpler and less space efficient, but does not require rule priorities, making it deployable on our switches. It uses a simple heuristic that attempts to wildcard the source address of rules at each switch. The algorithm finds the set of rules that share the same destination, TOS field, limiter, and output port. If the source address is identical for the entire set, the optimizer will collapse them into one rule with a wild card for the source address. Once this optimization occurs, the Blender controller must not admit any future rules that match this wildcard. We use this optimizer in our evaluation.

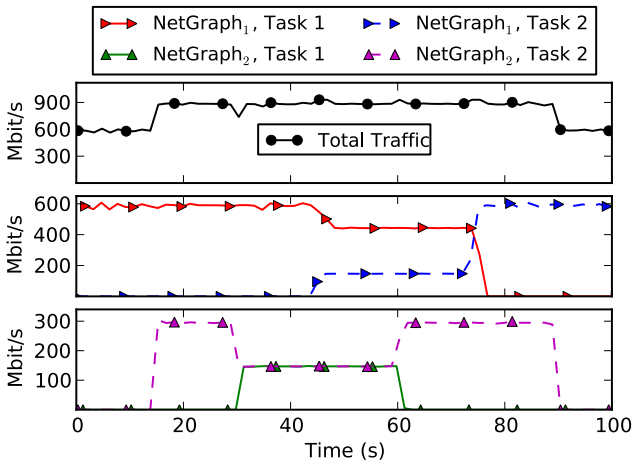

Figure 2: A composition of multiple isolation upgrades over a shared physical path. The three figures show total utilization, traffic for a 600-Mbps network slice, and traffic for a 300-Mbps network slice, from top to bottom, respectively. Each slice's events do not impact the tenant tasks operating in the other.

7. EVALUATION

Our evaluation explores Blender's ability to mix multiple upgrades across a shared network. These experiments demonstrate the functionality of the sample upgrades described in Section 4 and allow us to quantify Blender's network resource requirements.

7.1 Physical testbed

We explore Blender using a three-level, $k = 6$ fat tree [1] built from six 48-port HP ProCurve 6600-series switches running the K.14.87o OpenFlow firmware. The switches can store 1500 rules and contain 256 hardware rate policers. We use VLANs to divide each physical switch into eight six-port mini-switches. Note that while our prototype runs across a fat tree topology, we limit our topology to a single core mini-switch to reproduce topologies used by Oktopus [5] and Seawall [33].

The switching fabric carries traffic for 50 hosts, each of which contains an Intel Xeon X3210 and 4 GB of main memory. Every host uses two gigabit Ethernet interfaces; the first connects to a control network for system administration and interfacing with the Blender controller, and the second interface connects to one of the edge-level, Blender-enabled mini-switches. The hosts each house four Linux-KVM virtual machines, totaling 200 VMs.

7.2 Performance isolation

Hierarchical tenant allocation. We demonstrate Blender's ability to combine multiple isolation models on a shared physical network by hierarchically composing upgrades, as described in Section 3.1. This experiment uses the Squid upgrade to isolate pairs of Jetty-allocated tenants from one another. We first use Squid to create two virtual networks of fixed capacity: 600 Mbps (NetGraph$_1$) and 300 Mbps (NetGraph$_2$). Within each virtual network, we allocate two tenants using the Jetty upgrade. We have engineered the tenant requests to ensure that all six share a common path in the core of the physical network[4]. We then start traffic flows within the four Jetty-allocated tenants at various times.

Figure 2 shows the rate utilization across the shared physical path. The three sub-figures show total utilization, traffic for

[4]This was done by setting C_{VMs} to control the placement of VMs.

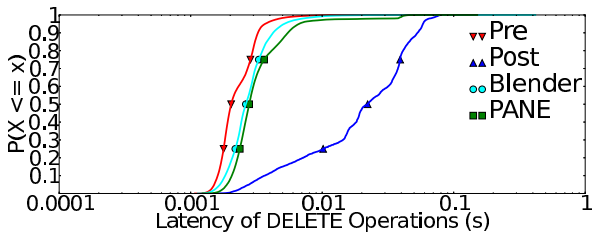

Figure 3: A comparison of performance isolation while running Zookeeper as provided by Blender and PANE [10]. Blender similarly achieves low latency without requiring modifications to Zookeeper.

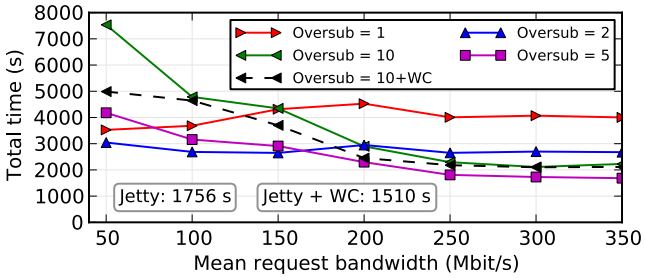

Figure 4: The time it takes Jetty and Squid to complete a set of 100 tenant tasks as we vary the tenant request parameters.

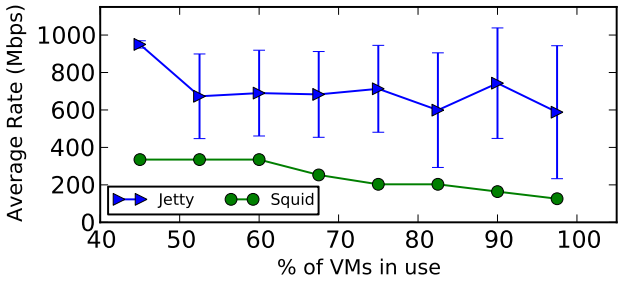

Figure 5: The mean and standard deviation (bars) of the bandwidth between the VMs of a 15-node experimental task with a varying number of background tasks.

NetGraph$_1$, and traffic for NetGraph$_2$, from top to bottom, respectively. Initially, a single tenant, operating within a Jetty virtual network with a weight 3, is executing a task on NetGraph$_1$. At time 15, a tenant operating within a Jetty virtual network with weight 1 begins executing a task on NetGraph$_2$. Fifteen seconds later, a second Jetty-allocated tenant with weight 1 arrives on NetGraph$_2$, causing them to each share 50% of NetGraph$_2$'s 300 Mbps capacity. At time 45, a final Jetty-allocated tenant with weight 1 starts a task on NetGraph$_1$, triggering NetGraph$_1$'s capacity to be reallocated in a 3:1 ratio. At subsequent 15-second intervals, one of the remaining tenants departs.

Notice that each pair of Jetty tenant tasks performs as if they were on their own network, responding only to new tenant arrivals on their Squid virtual network. Moreover, the combined traffic never exceeds the sum of the capacities of the Squid virtual networks.

PANE comparison. To measure the effect of Blender's performance isolation upgrades on latency, we reproduce an experiment performed by PANE [10] in which a client makes repeated DELETE requests to five Apache Zookeeper servers in the presence of heavy background traffic (iperf). The experiment runs four scenarios, and Figure 3 displays the results. The baseline, *Pre*, executes Zookeeper without background traffic, while*Post* represents the worst-case scenario with no isolation. The remaining two scenarios employ either PANE or Blender to isolate Zookeeper operations.

PANE requires modifications to Zookeeper for it to take advantage of their participatory APIs. For the details of PANE's behavior see [10]. For Blender, we represent Zookeeper and the background traffic as separate tenants, using the Squid upgrade to reserve 100 Mbps per host for Zookeeper. The background traffic receives the remaining network capacity. Using Blender, Zookeeper achieves comparable latency without application modifications.

7.3 Tenant throughput

Next, we evaluate the time for Blender to complete a set of 100 tenant tasks, using both Jetty and Squid, in a similar fashion to the evaluation in [5]. Each of the 100 tenants request a number of VMs drawn from an exponential distribution with a mean of 18. Each VM in the tenant's virtual network chooses one destination among the other VMs and sends 1500 MB to it. The tenant's task is considered complete when every VM has completed its transfer. A tenant dispatcher issues tenant requests Blender, and each requested tenant network is either accepted, in which case the tenant's task begins running, or it is rejected due to insufficient resources (lack of capacity or available VMs). The dispatcher continues to request any rejected tenants until they are all completed. We measure the time between the first tenant beginning and the final tenant ending.

For Jetty, each VM is equally likely to send to any other VM, and all tenants request equal weights for their paths. For Squid, the parameter selection process is similar to that in the evaluation of Oktopus [5]. Because Squid sub-divides its virtual networks into clusters, we assign VM destinations according to the oversubscription factor O such that the likelihood of a VM choosing an inter-cluster destination is $\frac{1}{O}$ for $O > 1$ and uniform likelihood for $O = 1$. Finally, Squid-allocated tenants draw their bandwidth request value from an exponential distribution whose mean we vary.

Figure 4 shows the time each configuration takes to complete the set of tenant tasks. We plot the completion time for Squid oversubscription factors of 1, 2, 5, 10, and 10 with work conservation enabled. We also show Jetty with and without work conservation. Jetty has no notion of bandwidth or oversubscription, so we present Jetty's completion times separately on the plotted results. Note that Blender allowed us to seamlessly add work conservation to improve performance, even for an isolation model that did not originally include such functionality. Without a system like Blender, such a comparison would be difficult to perform.

7.4 Allocator bandwidth and variability

In the previous experiment, Jetty achieves a higher overall throughput than Squid. However, the increase in throughput comes at a cost with respect to Squid in the form of variance. To illustrate this effect, we adjust the load on the network by installing a varying number of background Jetty tasks. Next, we install one experimental task and measure the capacity between all pairs of its VMs. Figure 5 plots the mean and standard deviation of the experimental task as we vary the total number of VMs in use. As expected, provisioning the experimental task with the Jetty allocator yields a higher average capacity than Squid, which has a

Figure 6: Rule and policer count, along with the number and average size of concurrently executing tenant tasks for a switch during the Squid $Mean = 150, Oversub = 5$ run described in Figure 4.

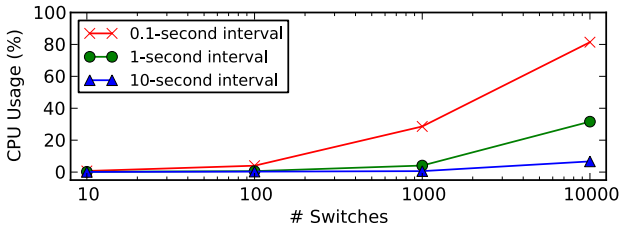

Figure 7: Work conservation's CPU utilization at the controller for a variety of switch counts and control loop intervals.

substantially lower variance. This experiment demonstrates that if given the opportunity to choose, different tenants would benefit from selecting an allocator that matches their applications' bandwidth and performance variability requirements. This experiment demonstrates that, if given the opportunity to choose, different tenants would benefit from choosing an allocator that best fits their applications' performance needs.

7.5 Hardware resources

Because it shares capacity evenly among concurrently executing tenants, Jetty is only constrained by the number of available VMs. As a result, it tends to keep network utilization high, and it performs relatively well, particularly with work conservation enabled. For low mean bandwidths, Squid tends to run out of VMs before it exhausts the available network capacity, leading to poor performance for larger oversubscription factors. Enabling work conservation provides a significant benefit in this region due to improved network utilization. As we increase the mean bandwidth, oversubscription becomes more beneficial due to capacity, rather than VMs, becoming the constraining resource. For the number of VMs in our network, an oversubscription factor of 5 appears to perform the best for Squid tenants.

Switches have a finite hardware capacity, which constrains their ability to store rules and police traffic. This affects the number of concurrent tenants Blender can support. This is a complex problem, as the number of rules and policers needed by a switch depends on tenant count, network size, network topology, and upgrade selection.

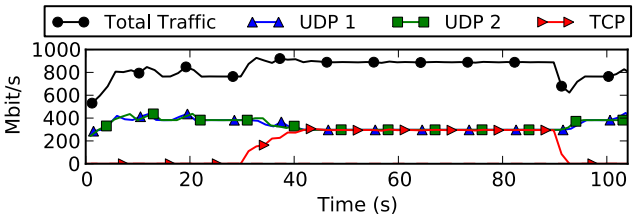

Figure 8: The work conservation upgrade controlling the traffic of three senders across a shared physical path. Each sender has a fixed-capacity reservation of 300 Mbps, which it contributes to work-conservation pool for a total of 900 Mbps. The work conservation upgrade divides the pool among the set of active senders, which varies over time.

Figure 6 illustrates the rule and policer counts for a single switch, along with the number of concurrently executing tenants and their average task size, during the execution of the Squid $Mean = 150, Oversub = 5$ run depicted in Figure 4. We selected the switch with the maximum rule count during the run. The other switches followed a similar pattern. As we described in Section 6.3, we use a sub-optimal rule optimizer due to our switches not implementing rule priority. With our better optimizer, the rule count would be equal to that of the policer count, which is a significant decrease.

Over the course of the experiment, we do not prescribe the order in which tenants execute. Our dispatcher requests tenant networks one at a time, cycling though all outstanding tenants until it finds one for which the system has sufficient resources. As the lower graph indicates, this scheme tends to initially execute smaller tenants, as they are more likely to find the resources they desire as tenants enter and leave. Around the time the 60th tenant begins executing, the switch reaches a peak of 65 allocated policers, which corresponds to the maximum concurrently-executing tenant count of 32. As the system shifts to larger task sizes, the concurrent tenant count decreases. This leads to an inflection point in the graph, beyond which policer usage decreases and rule usage increases. The trends for Jetty are similar to that of Squid, though less pronounced, due to Jetty being less constrained by available capacity.

Another important resource in any SDN is the controller. Blender allows for upgrades to execute code on the controller in response to dynamic network conditions. While this may sound like it would tax the controller, the upgrade programming model naturally enables extensions that are light-weight, with a tunable control loop interval to trade off accuracy and resource usage. As an example, our work conservation extension executes a control loop in which it periodically requests switch statistics and reassigns rate limits. Figure 7 shows the processing requirement of the work conservation extension at the controller, varying the control loop interval and the number of switches.

7.6 Dynamic upgrades

To exhibit the functionality of our sample dynamic upgrades (Section 4.3), we evaluate several upgrades:

Work conservation. To demonstrate work conservation, we configure and install virtual networks for three tenants, each of which reserves 300 Mbps across a shared path and contributes its reservation to a work-conserving pool. Figure 8 depicts the experiment's result. Initially, two of the tenants send UDP datagrams as quickly as possible across the shared path. Due to the third tenant being idle, the UDP senders draw from the excess work-conservation

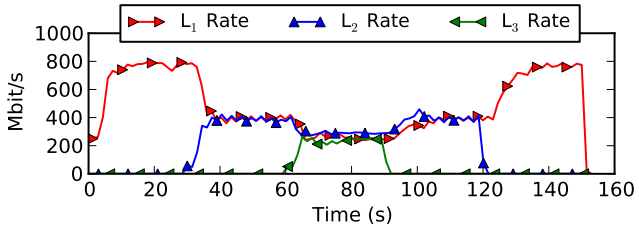

Figure 9: The Distributed Rate Limiting (DRL) upgrade enforcing an 800-Mbps global rate limit across three physically-disjoint network paths. The plot shows the outgoing rates of the three limiters, L_1, ..., L_3, that are responsible for policing the three paths. As senders begin and end their transmissions, the total traffic rate remains below the global 800-Mbps limit.

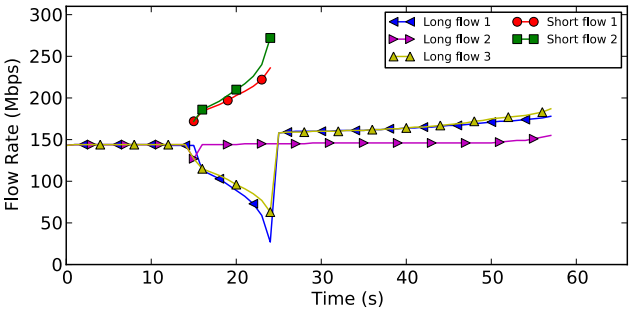

Figure 10: A combination of the BMA flow assignment upgrade and a deadline-aware flow scheduling upgrade. Long flows send one gigabyte of data with a deadline of 60 seconds, and short flows send 200 megabytes with a deadline of 10 seconds.

pool and each send at just over 400 Mbps. After 30 seconds, the third tenant begins sending TCP traffic over the shared path. Despite competing with UDP, the work conservation upgrade ensures that the third tenant's TCP traffic receives its 300-Mbps reservation. At 90 seconds, the TCP flow completes, and the UDP senders are again given the excess pool capacity.

Distributed rate limiting. To test our DRL upgrade, we construct a virtual network in which a set of three sending VMs transmit to three receiving VMs across three fully-disjoint paths. For each path, the sender creates a fixed 900-Mbps reservation along the path to its corresponding receiver. We then configure the edge-switches of all three senders to execute the DRL upgrade with a global rate limit of 800 Mbps.

Figure 9 shows the traffic departure rates from the edge-switch rate limiters associated with the traffic for each of the three senders. Every 30 seconds, one sender begins or ends its transmission, causing the DRL upgrade to reallocate the 800-Mbps budget amongst the active senders. Without DRL, we would see each sender fully utilizing its 900-Mbps reservation, however with DRL enabled, the aggregate sending rate of all three senders remains below the global limit.

Flow path selection and deadline-awareness. To evaluate the composition of these two upgrades, we configure a tenant with three paths between two VMs within a pod in our testbed. Each path is configured with a 300-Mbps reservation, and initially, the sending VM transmits three TCP flows to a receiving VM across three parallel paths. We use the BMA flow placement upgrade to spread the flows across the available paths. Our traffic generator aims to send one gigabyte of data over each of the three flows with a deadline of 60 seconds. After 15 seconds, two additional flows begin, each with a size of 200 megabytes and a deadline of 10 seconds.

Figure 10 displays the results. Prior to the additional flows beginning, the three original flows converge to the necessary rate for on-time completion. When the two short-deadline flows start, our upgrade displaces two of the longer flows until they complete. Afterwards, the two displaced flows converge on a higher rate to make up for their displaced time. The third flow, which never shared a link with either of the short-deadline flows, continues at its original rate.

8. CONCLUSION

This work introduced Blender, a framework that enables data center operators to author and compose modular upgrades on a shared network infrastructure. Not only can Blender express many recently proposed network performance isolation models, but it allows them to easily compose with additional functionality. Data center operators can differentiate their cloud network offerings by offering tenants new upgrades, without requiring tenants to modify their deployed code base. In addition, our experiments indicate that enforcement of tenant- based resource allocation is practical given current rate limiter facilities in modern top of rack switches.

Acknowledgements

Portions of this work were funded by the UC San Diego Center for Network Systems; grants from the Broadcom Foundation, Cisco, Ericsson, and Google; and the National Science Foundation through CNS-0917339 and CSR-1018808.

9. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[3] 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 *NSDI*, 2012.

[4] Amazon. Elastic Load Balancing. `http://aws.amazon.com/elasticloadbalancing`.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[6] T. Benson and A. Akella. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *SOCC*, 2011.

[7] T. Benson, A. Akella, and D. A. Maltz. Network Trafic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[8] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[9] Cisco. *Quality of Service on the Cisco Catalyst 4500 Classic Supervisor Engines*, 2006.

[10] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[12] 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 *SIGCOMM*, 2009.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 2008.

[14] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Z. Yongguang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[15] T. Hoff. Latency is Everywhere and it Costs You Sales, July 2009. `http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html`.

[16] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. IETF RFC 2992, 2000.

[17] 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 *OSDI*, 2010.

[18] V. T. Lam, S. Radhakrishnan, A. Vahdat, G. Varghese, and R. Pan. NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM CCR*, 42(3), 2012.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 2008.

[20] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *INFOCOM*, 2010.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *NSDI*, 2013.

[22] J. Mudigonda, P. Yalagandula, B. Stiekes, J. Mogul, and Y. Pouffary. Netlord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[23] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.

[24] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[25] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[26] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[28] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*, 2011.

[29] S. Rooney, J. E. van der Merwe, S. A. Crosby, and I. M. Leslie. The Tempest: A Framework for Safe, Resource-Assured, Programmable Networks. *IEEE Communications*, October 1998.

[30] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[31] 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 *SIGCOMM*, 2012.

[32] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[33] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[34] A. Voellmy, A. Agarwal, and P. Hudak. Nettle: Functional Reactive Programming for OpenFlow Networks. Technical Report YALEU/DCS/RR-1431, Yale University, July 2010.

[35] G. Wang and T. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[36] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[37] D. Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *SOSP*, 1999.

[38] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Envrionments. In *OSDI*, 2008.