Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds

Aaron Gember[‡], Anand Krishnamurthy[‡], Saul St. John[‡], Robert Grandl[‡], Xiaoyang Gao[‡], Ashok Anand[†], Theophilus Benson[*], Vyas Sekar[°], Aditya Akella[‡]

[‡]University of Wisconsin – Madison, [†]Instart Logic, [*]Duke University, [°]Stony Brook University

Abstract

Enterprises want their in-cloud services to leverage the performance and security benefits that middleboxes offer in traditional deployments. Such virtualized deployments create new opportunities (e.g., flexible scaling) as well as new challenges (e.g., dynamics, multiplexing) for middlebox management tasks such as service composition and provisioning. Unfortunately, enterprises lack systematic tools to efficiently compose and provision in-the-cloud middleboxes and thus fall short of achieving the benefits that cloud-based deployments can offer. To this end, we present the design and implementation of Stratos, an orchestration layer for virtual middleboxes. Stratos provides efficient and correct composition in the presence of dynamic scaling via software-defined networking mechanisms. It ensures efficient and scalable provisioning by combining middlebox-specific traffic engineering, placement, and horizontal scaling strategies. We demonstrate the effectiveness of Stratos using an experimental prototype testbed and large-scale simulations.

1. Introduction

Surveys show that enterprises rely heavily on in-network middleboxes (MBoxes) such as load balancers, intrusion prevention systems, and WAN optimizers to ensure application security and to improve performance [45, 46]. As enterprises move their applications and services to the cloud, they would naturally like to realize these MBox-provided performance and security benefits in the cloud as well. Recent industry trends further confirm this transition with an increasing number of virtual network appliances [2, 11, 13], and in-the-cloud network services [1, 4, 6, 7, 9].

At a high level, virtualized MBox deployments create new challenges as well as new opportunities for MBox *composition* (also referred to as service chaining [42]) and *provisioning* to meet desired performance objectives. The proprietary and non-deterministic nature of processing make these tasks hard even in traditional network deployments [30, 41]—the *dynamic*, *virtualized*, and *multiplexed* nature of cloud deployments compound the problem, leading to brittleness, inefficiency and poor scalability (§2).

MBox Composition Enterprises often need to chain multiple MBoxes together; e.g., traffic from a gateway must pass through a firewall, caching proxy, and intrusion prevention system (IPS) before reaching an application server. Today, such policy is implemented by physically wiring the topology. Virtualization offers a new opportunity to break this coupling between the policy and topology. At the same time, the chaining logic must now be implemented *via forwarding mechanisms*, which raises unique challenges in the face of dynamic changes to MBox chains. In particular, the stateful nature of MBoxes coupled with the complexity of packet *mangling* operations they perform (e.g., NATs rewrite headers and proxies terminate sessions) makes it difficult to ensure forwarding correctness and efficiency.

MBox Provisioning Traditional MBox deployments are typically overprovisioned or require drops in functionality in the presence of load; e.g., an IDS may disable DPI capabilities under load [12]. Virtualized cloud deployments offer the ability to flexibly scale MBox deployments as needs change. At the same time, the heterogeneity in MBox processing, characteristics of MBox workloads, and multiplexed nature of cloud deployments makes it challenging to address resource bottlenecks in an efficient and scalable manner. Furthermore, poor network placement or routing may introduce network-level effects that may adversely impact provisioning decisions by causing needless scaling.

While many MBox vendors are already making virtual MBoxes readily available to enable enterprises to deploy in-cloud MBoxes, there's a dearth of systematic tools to address the above composition and provisioning challenges. To this end, we design and implement, Stratos, *a new network-aware orchestration layer for virtual MBoxes in clouds*.

Stratos provides a novel, flexible software-defined networking (SDN) solution to the composition problem that leverages the virtualized nature of the deployment. In contrast to prior SDN solutions that require expensive and potentially inaccurate in-controller correlation or changes to MBoxes [26, 41], Stratos engineers a simpler solution by marginally over-provisioning an MBox chain to explicitly avoid potential steering ambiguity in the presence of mangling MBoxes (§4).

To ensure efficient and scalable provisioning, Stratos employs a scalable multi-level approach that carefully synthesizes ideas from traffic engineering [28], network-aware virtual machine placement [20, 49], and elastic compute scaling [10] (§5). As a first and light-weight step, it uses a flow

distribution mechanism to address transient compute or network bottlenecks, without having to know the nature of the bottleneck itself. When this is insufficient, Stratos locates persistent compute or network bottlenecks, and applies progressively heavier techniques via network-aware horizontal scaling and migration. The techniques ensure that the network footprint of MBox chains is low, and compute resource utilization is maximized meeting our efficiency goals.

We have implemented a fully featured Stratos prototype (≈12K lines of Java code), including a forwarding controller written as a FloodLight module [5] and a stand-alone resource controller. We evaluate this prototype in a 36 machine testbed using a variety of MBox chains and synthetic request workloads. We also simulate Stratos to understand its properties at larger scale. We find that our composition mechanisms impose a 1ms overhead on the completion time per flow for each mangling MBox included in a chain. By construction, Stratos always maintains correct composition, whereas state-of-the-art techniques have ≈19% error rate in the presence of mangling and dynamic provisioning [41]. Our provisioning mechanisms satisfy application objectives using up to one-third fewer resources and invoking up to one-third fewer heavy-weight operations for the scenarios we consider. Last, we show that Stratos's controllers can perform upwards of 50 provisioning operations per second; given that provisioning occurs on the order of tens of seconds, this is sufficient to support hundreds of tenants.

2. Requirements and Related Work

Our goal is to build a MBox orchestration system that enables cloud tenants to (1) *compose* rich, custom chains atop their MBox deployments. A chain is a sequence of MBoxes that process a given traffic subset: e.g., an enterprise may require traffic from a remote office to a company web server to pass through a firewall and caching proxy (chain 1), and traffic from home users to pass through a firewall, proxy, and intrusion prevention system (chain 2) (Figure 1); and (2) automatically *provision* a suitable amount of resources for each chain to optimally serve tenants' workloads.

We argue that such a system must simultaneously meet the following requirements:

- **Correctness:** The physical realization of chains must correctly apply high level policies to traffic sub-streams.
- **Application-specific objectives and efficiency**: The system should enable tenants to use the minimal amount of resources necessary to realize application-specific service-level objectives (SLOs).
- **Scalability:** The system should scale to hundreds-to-thousands of MBox chains from many tenants.

Meeting these requirements is challenging on four key fronts. (*i*) the *closed nature* of third-party MBoxes that makes it difficult to instrument them; (*ii*) *diversity*, both in the nature of actions applied to packet streams and in the amount of resources, such as CPU, memory, and network

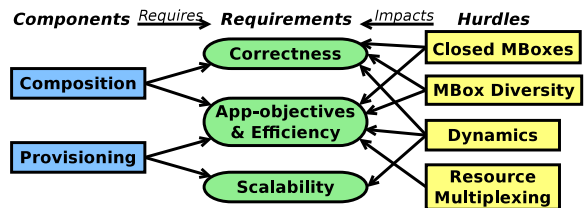**Figure 1: Example MBox deployment with two chains (shown in red/blue).**

Figure 2: Interaction between the requirements for composition and provisioning and the challenges imposed by middleboxes and cloud deployments

bandwidth, consumed during packet processing; (*iii*) the *shared nature* of cloud environments; and (*iv*) the *dynamicity* arising from tenant workload variations (and potential MBox migration).

Next, we explain how these factors make existing composition and provisioning solutions ineffective for meeting our requirements. In the interest of brevity, we highlight only salient aspects and summarize the interactions in Figure 2.

2.1 Composition

We first need a mechanism to enforce the appropriate steering of traffic subsets across a chain of MBoxes.

A steering mechanism must meet two high-level correctness requirements. First, at the granularity of an individual MBox, we note that many MBoxes are *stateful* and require both directions of TCP sessions for correct functionality. Thus, when a workload shift forces an *MBox deployment change* due to re-provisioning (§2.2), forwarding rules must be updated to preserve *flow affinity*. Second, at the granularity of an individual MBox chain, we need to ensure that a packet goes through the desired processing sequence. However, many MBoxes *mangle* packets by dynamically modifying packet fields; e.g., NATs rewrite headers and caching proxies terminate connections. Thus, the traffic steering rules must account for modifications to determine the next hop MBox for mangled packets.

Existing approaches to steering fail to address at least one of these requirements. Some techniques (e.g., PLayer [30] and SIMPLE [41]) are designed for *static* MBox deployments. As such, they lack a nuanced approach for adjusting the fraction of traffic assigned to specific MBox instances. They simply re-divide the flow space among the available instances and replace existing steering rules; this may cause some currently active flows to switch instances and violate the affinity requirement (Table 1). Other techniques (e.g.,

Framework	Handles Mangling	Maintains Affinity	No MBox Changes	Minimal Rules
PLayer [30]	✓	X	✓	?
SIMPLE [41]	≈	X	✓	✓
Consistent [43, 48]	X	✓	✓	✓
Per-Flow Rules	X	✓	✓	X
FlowTags [26]	✓	?	X	✓
SC Header [21, 42]	✓	?	X	?

Table 1: Comparison of existing approaches for middle-box composition.

consistent updates [43, 48] or using per-flow rules [22][1] may be able to ensure that active flows maintain affinity; however, they do not account for mangling.

Tackling the mangling problem is specially difficult because MBoxes are closed and proprietary. Existing solutions fall into two categories: (*i*) use flow correlations to reverse-engineer the mangling relationships (e.g., SIMPLE [41]), or (*ii*) require MBoxes to add persistent network handles to packets (e.g., FlowTags [26] and service chaining headers [21, 42]). The former is both expensive (requiring multiple packets to be sent to the controller) and error-prone (e.g., SIMPLE has 19% error), while the latter needs MBox modifications (Table 1).

Efficiency implies minimal memory footprint in the switches and low latency overhead in the controller that manages switch forwarding rules. Existing solutions to this issue [48, 50] apply exclusively to simple routing and load balancing scenarios, and cannot accommodate mangling MBoxes and ensure affinity. As such, we need new schemes.

As we will see in §4, Stratos leverages the virtualized deployment to engineer simpler more efficient solutions for both stateful forwarding and to handle mangling MBoxes.

2.2 Provisioning

Two related issues must be addressed to ensure that tenant applications meet their service-level objectives: (*i*) **detection** to determine where a resource bottleneck (or excess) exists in an MBox chain, and (*ii*) **provisioning** decisions on how/where resources need to be added (or removed) for the chain. At a high-level, existing approaches for detection can lead to inefficiency in the MBox context, and existing provisioning mechanisms can cause both inefficiency and scalability issues.

2.2.1 Resource bottleneck detection

A common approach (e.g., RightScale [10]) is to monitor CPU and memory consumption on individual virtual machines (VMs) and launch additional VMs when some critical threshold is crossed. Unfortunately, the shared nature of clouds, and the unique and diverse resource profiles of MBoxes together cause this approach to both miss bottle-

necks (impacting applications) and incorrectly infer bottlenecks (impacting efficiency). For instance, running multiple virtual MBoxes on the same host machine can lead to a memory cache bottleneck [23]. Unfortunately, existing approaches will not lead to an appropriate scale-out response in such cases, impacting application performance. Similarly, MBoxes that use polling to retrieve packets from the NIC will appear to consume 100% of the CPU regardless of the current traffic volume [24]. In such cases, existing approaches will cause spurious scale-out and reduce efficiency.

Furthermore, these approaches do not consider network-level effects, which can lead to further impact on applications and inefficiency. Indeed, MBox deployments in clouds are often impacted by transient network problems that may cause application flows traversing MBoxes to get backlogged and/or timed-out [40]. Persistent network hotspots [19] can have a similar effect. Thus, we may incorrectly conclude that the bottlenecks lie at MBoxes experiencing backlogs leading to ineffective scale-out response.

2.2.2 Provisioning decisions

In a general setting, we can use one of three options to alleviate bottlenecks: (1) *horizontal scaling* to launch more instances based on resource consumption estimates [25]; (2) *migrate* instances to less loaded physical machines [49]; and (3) choose appropriate *placement* to avoid congested links (e.g., [18, 34, 36, 47]). Unfortunately, the dynamicity of clouds coupled with the varying resource consumption of some MBoxes renders existing techniques, and combinations thereof, inefficient and/or not scalable.

For instance, MBox resource consumption is quite diverse and workload dependent [27]. Thus, scaling based on specific resource indices may be inefficient and may not really improve end-to-end application performance. Similarly, the bandwidth consumption of MBoxes can vary with the traffic mix (e.g., the volume of traffic emitted by a WAN optimizer depends on how much the traffic is compressed [15]) and data center workloads can vary on the order of tens of seconds to minutes [19], so placement decisions may only be optimal for a short period of time. Frequently invoking migration or scaling to accommodate these changes will result in over-allocation of compute resources. It will also introduce significant management overhead (for VM provisioning, virtual network set up, etc.) that can limit system scalability.

Thus, we need a systematic framework for resource provisioning that ensures MBox chain efficiency and system scalability in the face of MBox and cloud dynamics.

3. Stratos Overview

Figure 3 shows an overview of the Stratos system with the interfaces between the different components that we discuss briefly below. Stratos is a network-aware orchestration layer for virtual MBoxes in clouds. We synthesize novel ideas with
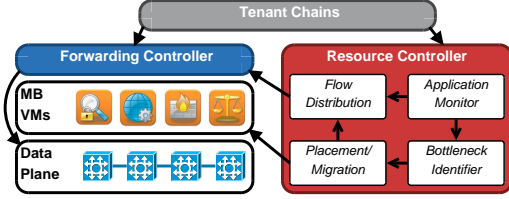
---

[1] When a new flow starts, a central controller installs flow-specific rules along the entire path for an MBox chain.

Figure 3: Stratos overview

existing mechanisms to meet the provisioning and composition requirements discussed in §2.

Tenant Input. Tenants provide a logical specification of their MBox chains, including the applications/users associated with each chain and the VM images to use for each MBox.

Forwarding Controller. The data plane, composed of virtual switches and tunnels, is programmed to forward traffic through MBox instances according to chain specifications and computed flow distributions. Stratos pre-processes the input chains to handle packet mangling. It carefully manages individual flows uses a combination of tag-based and per-flow rules to ensure correctness and efficiency (§4). The forwarding controller also receives new provisioning strategies output by the resource controller (see below), and updates the data plane configuration to ensure correct composition during and after these dynamic provisioning decisions.

Resource Controller. Stratos uses end-to-end application performance as a common, resource-agnostic indicator of an MBox chain facing performance bottlenecks. It monitors application performance and receives resource statistics from the individual MBox and application VMs as well as network utilization statistics. Note that cloud providers already provide extensive APIs to export this monitoring information to tenants [1]. (As such, the design of such a monitoring infrastructure is outside the scope of this paper.)

It uses a combination of three mechanisms—flow distribution, horizontal scaling, and instance migration—applied at progressively coarser time-scales to identify and address bottlenecks in an efficient and scalable manner (§5). Intuitively, flow distribution rebalances the load at fine timescales to MBox replicas to address transient compute and network bottlenecks. This is lightweight and can be applied often and in parallel across many chains, aiding control plane scalability. Horizontal scaling eliminates persistent compute bottlenecks. Congestion-aware instance migration and horizontal scaling address persistent network bottlenecks. In both cases, scaled/migrated instances are provisioned in a network-aware fashion, and flow distribution is applied to improve efficiency.

In the next two sections, we present the detailed design of the Stratos forwarding and resource controllers.

4. Stratos Forwarding Plane

Our key insight to overcome the challenges discussed in §2.1 is that we can leverage unique features of the virtualized environment to engineer efficient composition approaches that handle mangling middleboxes and maintain affinity.

4.1 Addressing MBox Mangling

As discussed in §2, mangling/connection terminating (M/CT) MBoxes interfere with the ability to correctly forward packets to the appropriate downstream MBoxes. First, the identifiers required for selecting the appropriate sequence of downstream MBoxes may be obscured by an M/CT MBox's packet modifications. Second, flow forwarding rules set up in switches downstream from an M/CT MBox will cease to be valid when packet headers change.

Stratos addresses the former issue by identifying potential sources of forwarding ambiguity in the set of logical chains C provided by a tenant and applying a correctness-preserving transformation to generate a logically equivalent C' that is used for all subsequent forwarding decisions. The latter is addressed by logically dividing a chain c into subchains and installing per-flow forwarding rules for a particular subchain when the first packet of a flow is emitted by the first MBox in that subchain.

Prior to applying either mechanism, Stratos must identify the set of MBoxes in the C that are potential M/CT MBoxes using either: (1) operators' domain expertise[2], or (2) monitoring the ingress/egress traffic of each MBox and checking if the output packets fall in an expected region of the flow header space [31].

Correctness-Preserving Transformation. Given the set of chains C, we can create a logical graph $G = \langle V, E \rangle$, where each $v \in V$ is an MBox in a tenant specified logical deployment and an edge $m \to m'$ exists if there is a chain c with the corresponding sequence. For each M/CT MBox m, we exhaustively enumerate all downstream paths in the graph G starting from m; let this be $downstream_m$. Then, we create $|downstream_m|$ *clones* of m with the clone m_i solely responsible for the ith path in $downstream_m$. For example, in Figure 4(a), there are two downstream paths from the proxy; thus we create two copies of the proxy. The intuition here is that the appropriate path i for a packet emitted by m may be ambiguous due to packet changes made by m; by explicitly allocating isolated instances m_i for each path i we avoid any confusion due to mangling. We rewrite each affected chain $c \in C$ with the corresponding clone m_i instead of m, and add the rewritten chain to C'.

We acknowledge that this transformation potentially increases the number of MBox instances that need to be used: e.g., even if one proxy instance was sufficient to handle the traffic load for both chains, we would still need two clones—

---

[2] While prior work [30] required a detailed model of M/CT MBoxes' mangling behavior, Stratos only requires operators to identify which MBoxes are M/CT MBoxes.
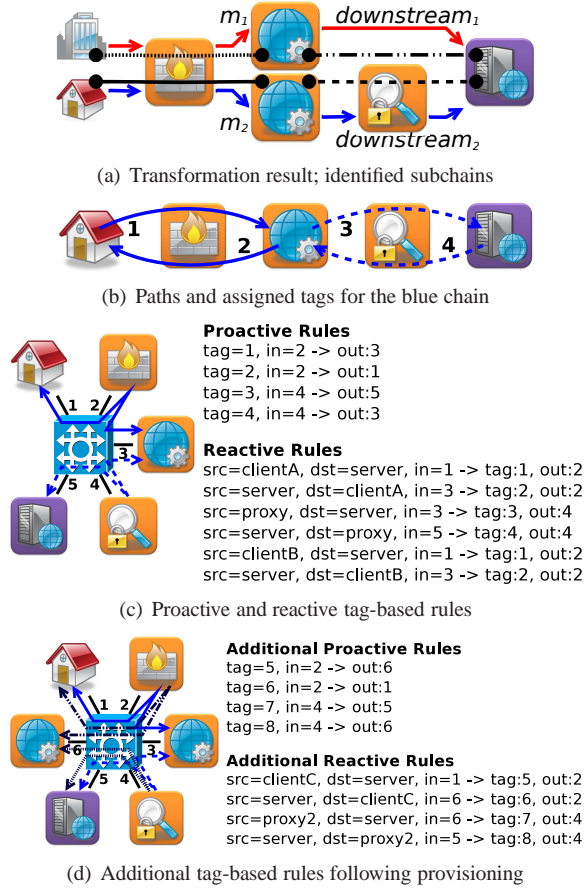
(a) Transformation result; identified subchains

(b) Paths and assigned tags for the blue chain

Proactive Rules
tag=1, in=2 -> out:3
tag=2, in=2 -> out:1
tag=3, in=4 -> out:5
tag=4, in=4 -> out:3

Reactive Rules
src=clientA, dst=server, in=1 -> tag:1, out:2
src=server, dst=clientA, in=3 -> tag:2, out:2
src=proxy, dst=server, in=3 -> tag:3, out:4
src=server, dst=proxy, in=5 -> tag:4, out:4
src=clientB, dst=server, in=1 -> tag:1, out:2
src=server, dst=clientB, in=3 -> tag:2, out:2

(c) Proactive and reactive tag-based rules

Additional Proactive Rules
tag=5, in=2 -> out:6
tag=6, in=2 -> out:1
tag=7, in=4 -> out:5
tag=8, in=4 -> out:6

Additional Reactive Rules
src=clientC, dst=server, in=1 -> tag:5, out:2
src=server, dst=clientC, in=6 -> tag:6, out:2
src=proxy2, dst=server, in=6 -> tag:7, out:4
src=server, dst=proxy2, in=5 -> tag:8, out:4

(d) Additional tag-based rules following provisioning

Figure 4: Steps for forwarding plane setup

that are then provisioned independently—to ensure correct composition. We believe that the simplicity and correctness guarantees made without modifying MBoxes, in contrast to prior solutions [26, 41], makes this tradeoff worthwhile.

Setting up forwarding rules. Given C' and knowledge of M/CT MBoxes, we logically split each chain into one or more *subchains*, with M/CT MBoxes delineating the subchains: e.g., the black lines in Figure 4(a) indicate subchains. Conceptually, a subchain represents a logical segment where the packet traverses the network without modifications to packet header fields that uniquely identify a flow.

Stratos needs to reactively set up flow rules when a packet for a new flow is emitted by the first MBox (or client/server) in a subchain. The Stratos forwarding controller chooses one of the possible *instance paths* that implement this specific subchain—i.e., an instance path contains a particular instance of each MBox in the subchain. (The specific path will be chosen using weighted round-robin with the weights determined by Stratos' flow distribution module described in §5.1.) The Stratos forwarding controller installs *exact match rules* in the virtual switches to which the MBox (and client/server) instances in the path are connected; the virtual switches themselves are connected by tunnels. Stratos also

installs flow rules for the reverse flow in order to maintain the flow affinity.

4.2 Maintaining Efficiency and Scalability

The above approach guarantees correctness in the face of mangling. However, we can further optimize rule installation both in terms of the number of rules required and the imposed load on the controller. The main insight here is that we can proactively install forwarding rules in virtual switches to forward traffic *within each subchain* using rules that forward on the basis of *tags* included in packet headers. Tags are set at the first instance of each subchain by reactive rules, as described above.

Using tag-based rules (versus per-flow rules) for forwarding within subchains reduces the total number of rules installed in virtual switches, leading to faster matching and forwarding of packets [37]. Additionally, proactively installing some rules reduces the number of rules the Stratos forwarding controller must install when new flows arrive, enabling fast forwarding and controller scalability. We quantitatively show these performance benefits in §7.2.

Initially there is only one instance of each MBox in a subchain, and thus only one possible instance path. We assign two tags to this path, one for the forward direction and one for the reverse: e.g., Figure 4(b) shows the paths and tags for the two subchains associated with the blue chain. A tag should uniquely identify both a subchain and a direction, so each tenant has a single tagspace.

Stratos installs wildcard rules that match both the forward (or reverse) tag and the virtual switch port of the prior MBox instance and output packets to the virtual switch port for the next MBox instance on the forward (or reverse) path: e.g., Figure 4(c) shows the rules for the blue chain. Per-flow rules are reactively installed as described in §4.1, but only at the first element in each sub-chain on the forward and reverse paths.

In our prototype, we place tags in the type-of-service field in the IP header, limiting us to 64 unique paths across all of a tenant's subchains; recent IETF drafts suggest adding a special header in which such a tag could be stored [21, 42].[3]

4.3 Affinity in Face of Dynamics

As additional MBox instances are provisioned, there become more possible paths for a subchain. Stratos allocates new forward and reverse tags just for the new paths, and installs the corresponding rules. The tags and forwarding rules for existing paths remain unchanged to ensure all packets of a flow traverse the same set of MBox instances in both directions; this is important for ensuring stateful MBoxes operate correctly. Figure 4(d) shows the additional rules that would be installed if the proxy was horizontally scaled; the rules shown in Figure 4(c) remain untouched.

---

[3] We assume the MBoxes do not modify these tag header bits, otherwise they would be treated as mangling MBoxes.

5. Provisioning

Stratos' network-aware control plane closely manages chain performance to satisfy application SLOs. In order to balance the two requirements of *efficiency* (i.e., use minimal resources for each MBox chain while meeting application objectives) and scalability (i.e., time/overhead of reconfiguration), we use the following multi-stage approach (Figure 5):
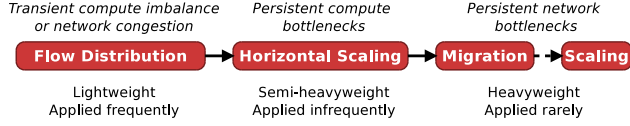

Figure 5: Process for detecting and addressing resource bottlenecks

1. To determine the existence of a bottleneck, Stratos leverages end-to-end application-specific metrics, as suggested by some prior approaches [17, 49]. Ultimately, these are the true indicators of whether the chain is in need of extra resources; they can detect chain performance issues arising from fine-grained workload variations.

2. Given that the most common situation is for MBox chains to face transient compute or network bottlenecks, we design a light-weight flow distribution scheme - that can be invoked often without impacting scalability - to address such bottlenecks. The scheme does not need to know the nature of the bottleneck to address it – whether compute or network, or even whether transient or persistent. It does not add new compute resources focusing instead on using them more efficiently (§5.1).
   When this scheme fails, we identify if this persistent is a compute or network bottleneck before taking appropriate measures.

3. We synthesize a suite of techniques to identify persistent compute bottlenecks. Each bottlenecked MBox is horizontally scaled by the minimal number of extra instances necessary (§5.2.1).

4. We use light-weight passive monitoring to identify persistent network bottlenecks. Since addressing such bottlenecks can be tricky, we use a multi-pronged approach to ensure scalability. We first attempt to migrate instances in an inverse congestion-sorted order. If this does not help, we horizontally scale instances affected most by network congestion (§5.2.2).

5. When #2 and #3 fail, we fall back to horizontally scaling all MBoxes by a fixed amount.
   In #3, #4, and #5 above, we directly monitor application level metrics to identify if our decisions were effective, and we employ network-aware placement and re-invoke flow distribution so that the scaled/migrated instances are used in the most efficient fashion.

Thus, our careful design systematically and accurately, addresses performance bottlenecks, and ensures efficient and scalable operation.

5.1 Flow Distribution

Flow distribution adjusts the fraction of flows assigned to a specific set of MBox instances so as to: (*i*) balance load on MBox instances to accommodate variations in the rates, sizes, and types of flows traversing the chain [25] and (*ii*) control transient network congestion from changing load on network links [19].

We cast this problem as a linear program (LP). Let c denote a specific chain and $|c|$ denote the number of MBoxes in that chain. The MBox (e.g., caching proxy, IPS) at position j in the chain c is denoted by $c[j]$. $M_{c[j]}$ is the set of instances of MBox $c[j]$; we use $i \in M_{c[j]}$ to specify that i is an instance of this MBox. V_c denotes the total volume (bytes) of traffic that must traverse chain c; we discuss in §5.4 how we determine the value of V_c.

Our goal is to split the traffic across the instances of each MBox such that: (a) the processing responsibilities are distributed roughly equally across them, and (b) the aggregate network footprint is minimized. In contrast with prior works that focus on equalizing MBox load [28, 41, 44], our formulation has the benefit of eliminating (or reducing) compute *and* network bottlenecks, and reducing the likelihood of them occurring in the future.

We need to determine how the traffic is routed between different MBox instances. Let $f(c, i, i^{'})$ denote the volume of traffic in chain c being routed from MBox instance i to MBox instance $i^{'}$. As a special case, $f(c, i)$ denotes traffic steered to the first MBox in a chain from a source element.[4]

$Cost(i \rightarrow i')$ denotes the network-level cost between two instances. In the simplest case, this is a binary variable— 1 if the two MBox instances are on machines in different racks and 0 otherwise. We capture current available bandwidth: Let r denote a specific rack, and $i \in r$ indicate that i is located in that rack. The current bandwidth available between two racks r and r' is denoted by $b(r, r')$. §5.4 describes how we measure available bandwidth in a scalable fashion.

LP Formulation. Figure 6 formalizes the flow distribution problem that Stratos solves. Eq (1) captures the network-wide footprint of routing traffic between instances of the jth MBox in a chain to the $j + 1^{th}$ MBox in that chain. For completeness, we consider all possible combinations of routing traffic from one instance to another. In practice, the optimization will prefer combinations that have low footprints.

Eq (2) models a byte conservation principle. For each chain and for each MBox in the chain, the volume of traffic entering the MBox has to be equal to the volume exiting it. However, since MBoxes may change the aggregate volume (e.g., a WAN optimizer may compress traffic), we consider a generalized notion of conservation that takes into account a gain/drop factor $\gamma(c, j)$: i.e., the ratio of ingress-to-egress

---

[4]For clarity, we focus only on the forward direction of the chain.

Minimize

$$\sum_{c} \sum_{j=1}^{|c|-1} \sum_{\substack{i,i' \ s.t \\ i \in M_{c[j]}; i' \in M_{c[j+1]}}} Cost(i, i') \times f(c, i, i') \quad (1)$$

subject to

$$\forall i, \forall c, \ s.t. \ i \in M_{c[j]} \ \& \ j > 1 :$$

$$\sum_{i' : i' \in M_{c[j-1]}} f(c, i', i) = \sum_{i' : i' \in M_{c[j+1]}} f(c, i, i') \times \gamma(c, j) \quad (2)$$

$$\forall c : \sum_{i : i \in M_{c[1]}} f(c, i) = V_c \quad (3)$$

$$\forall r, r' : \sum_{c} \sum_{\substack{i,i' \ s.t. \\ i \in r; i' \in r'}} f(c, i, i') \leq b(r, r') \quad (4)$$

$$\forall i : \sum_{c : i \in M_{c[j]}; j \neq 1} \sum_{i' : i' \in M_{c[j-1]}} f(c, i', i)$$

$$+ \sum_{c : i \in M_{c[1]}} f(c, i) \approx \sum_{c : i \in c; i \in M_{c[j]}} \frac{V_c}{|M_c[j]|} \times \Pi_{l=1}^{j} \gamma(c, l) \quad (5)$$

Figure 6: LP formulation for the flow distribution problem. The $\approx$ term in the last equation simply represents that we have some leeway in allowing the load to be within 10–20% of the mean.

traffic at the position j for the chain c. Stratos computes these ratios based on virtual switch port statistics (§5.4).

We also need to ensure that each chain's aggregate traffic will be processed; we also model this coverage constraint in Eq (3). We also need to ensure that total chain traffic across any two racks does not exceed the available bandwidth between the two racks; we model this bandwidth constraint in Eq (4). Finally, we use a general notion of load balancing where we can allow for some leeway; say within 10-20% of the targeted average load (Eq (5)).

5.2 Identifying and Addressing Bottlenecks

There are cases when flow distribution will be insufficient to improve end-to-end performance: e.g., when all instances of an MBox, or all paths to those instances, are heavily loaded, or when network/MBox loads are such than a redistribution is simply infeasible. In such cases, Stratos is forced to identify the type of bottleneck (compute or network) that exists and address it. To overcome the challenges outlined in §2, we adopt decouple the actions for dealing with the two types of bottlenecks: we focus on addressing compute bottlenecks first, followed by network bottlenecks.

5.2.1 Compute Bottlenecks

Stratos leverages a combination of host-level and per-packet metrics to determine whether a compute bottleneck exists, and for which MBoxes. Host-level metrics are used by existing scaling frameworks because these can be easily gathered from VMs [10, 49]. In addition to the host-level metrics, we rely on the packet processing time as it can cap-

ture *any* compute-related bottleneck, including CPU, memory space/bandwidth, cache contention [23], and disk or network I/O.[5]

Stratos declares an MBox instance to be bottlenecked if either: (*i*) average per-packet processing time increased by at least a factor δ over a time window, or (*ii*) CPU or memory utilization exceeds a threshold α and has increased by at least a factor β over a sliding time window.[6] We select these thresholds heuristically based on observing middlebox behaviors in controlled settings and varying the offered load. §5.4 discusses a scalable approach for gathering these metrics.

Horizontal Scaling. When compute bottlenecks are identified, Stratos horizontally scales each bottlenecked MBox and adds more such instances of. (Our current implementation increases only one instance at a time to avoid overprovisioning, but we could consider batched increments as well.) Crucially, these instances must be launched on machines that have, and likely will continue to have, high available bandwidth to instances of other MBoxes in the chain. This helps maximize the extent to which the new resources are utilized and minimizes the need to perform migration (which can hurt scalability) or further scaling (which can hurt efficiency) in the future.

We use a network-aware placement heuristic similar to CloudNaaS [20]: We try to place a new MBox instance in the same rack as instances of the neighboring MBoxes (or clients/servers) in the chain; if the racks are full, we try racks that are two hops away, before considering any rack. For each candidate rack, we calculate the flow distribution (§5.1) as if the new instance was placed there, and we choose the rack that results in the best objective value (i.e., minimizes the volume of inter-rack traffic).

A bottleneck at one MBox in a chain may mask bottlenecks at other MBoxes in the chain: e.g., a bottleneck at the proxy in Figure 1 will limit the load on the IPS; when the proxy bottleneck is resolved, load on the IPS will increase and it may become bottlenecked. Thus, we look for compute bottlenecks multiple times and perform scaling until no MBoxes in a chain exhibit the conditions discussed above.

5.2.2 Network Bottlenecks

Network bottlenecks may arise at any of the physical links that form the underlay for the virtual links (VLs) connecting neighboring MBox instances (i.e., the link from $i \in M_{c[j]}$ to $i' \in M_{c[j\pm1]}$). Using active probing to measure VLs' available bandwidth is not scalable: probing must occur (semi-)serially to avoid measurement interference caused by multiple VLs sharing the same underlying physical link(s). Hence, Stratos detects bottlenecked VLs by pas-

---

[5] This may not apply to MBoxes that do not follow a "one packet in, one packet out" convention (e.g., a WAN optimizer), and in these cases we can only use traditional CPU and memory utilization metrics.

[6] The increase factor avoids constant scaling of MBoxes which use polling.

sively monitoring the individual physical links that underly VLs. This requires gathering metrics from all physical network switches and identifying the physical links (e.g., using traceroute[7]) that form each VL; these tasks can easily be parallelized, as described in §5.4. A VL is bottlenecked when the minimum available bandwidth across the physical links that form the VL is less than a threshold δ.

Instance Migration. When a network bottleneck is identified, Stratos migrates affected MBox instances to less congested network locations. Since migrations are costly operations—involving, in our prototype, the instantiation of a new MBox instance with a more optimal placement and the termination of the instance affected by the network bottleneck(s)—performing the minimum number of migrations is crucial to maintaining system scalability. For this reason, Stratos first migrates MBox instances with the highest number of incident congested VLs and measures the migration's impact on end-to-end application performance before performing additional migrations. An MBox instance's new location is selected using the placement heuristic described in §5.2.1, with the added requirement that the available bandwidth between the MBox instance being migrated ($i \in M_{c[j]}$) and the instances of neighboring MBoxes in the chain ($i' \in M_{c[j\pm1]}$) must be greater than the current bandwidth consumed by i times some factor ρ.

In some cases, the network bottleneck(s) impacting an MBox instance may arise predominantly due to heavy traffic involving the instance itself (e.g., bandwidth needs may outstrip compute needs [27]). In these cases, and cases where all portions of the cloud network are congested, the network-aware placement routine will not yield a feasible solution. We address this situation by horizontally scaling the instance using the technique described in §5.2.1. This causes chain traffic to be spread among more MBox instances, reducing the network bandwidth needed by an individual instance and eliminating any network bottlenecks. It also causes under-utilization of compute resources on the affected instance(s), which reduces efficiency.

5.3 De-provisioning

To maintain efficiency, Stratos also eliminates excess compute resources. Excesses are identified by looking for MBox instances whose average per-packet processing time has dropped by at least a factor δ' over a time window. To avoid sudden violations of application SLOs, supposedly unneeded MBox instances are removed from service (but not yet destroyed) one at a time; flow distribution (§5.1) is invoked to rebalance load among the remaining instances. If the SLOs of *all* applications associated with the MBox chain are satisfied, then the instance is marked for permanent removal; otherwise, the instance is immediately restored to service (using the old flow distribution values). An instance

is fully destroyed only after all flows traversing it have finished (or timed-out).

5.4 Data Collection

The Stratos resource controller relies on many metrics when making provisioning decisions, but all of the metrics can be gathered in a scalable fashion by leveraging distributed monitoring agents and a centralized object store (e.g., [35]). Most applications already log end-to-end performance measures for other purposes; this can simply also be reported to Stratos. The volume of traffic traversing an MBox chain and the gain/drop factor and average per-packet processing time of MBoxes in the chain can be captured by querying the port statistics from each virtual switch. An agent running on each machine can perform such queries, as well as report VMs' CPU and memory utilization. Lastly, a collection of sensors can poll port statistics from physical switches using SNMP.

6. Implementation

We have implemented a full featured Stratos prototype consisting of several components (Figure 7). Stratos' modular design makes it easy to scale individual components as the number of tenants and the size of the cloud increases.

Forwarding Controller and Data Plane. The Stratos data plane is a configurable overlay network realized through packet encapsulation and SDN-enabled software switches. Each machine runs an Open vSwitch [8] bridge to which the virtual NICs for the VMs running on the machine are connected. The *network manager* establishes a full mesh of VXLAN tunnels for each tenant, connecting the vSwitches to which a tenant's VMs are connected.

The forwarding controller is implemented as a module ($\approx$2400 lines of Java code) running atop the Floodlight OpenFlow Controller [5]. Floodlight handles communication with the vSwitches, and the forwarding module interfaces with the resource controller and network manager using Java Remote Method Invocation (RMI).

Resource Controller. The *chain manager* ($\approx$6000 lines of Java) forms the core of the resource controller. It monitors the performance of tenant applications (through queries to the *metric datastore*) and executes the provisioning process (Figure 5) when SLO thresholds are crossed. Flow distribution is computed using CPLEX [3] and applied by the *forwarding controller*; scaling and migration decisions are applied by the *placement manager* ($\approx$ 3300 lines of Java). When placement decisions are made, the *compute manager* communicates with the Xen [14] hypervisor to launch VMs.

The metrics required for provisioning decisions reside in the *metric datastore*, currently a simple in-memory data store written in Java. A *VM monitor* runs on each machine and reports the CPU, memory, and network metrics for running VMs based on output from Xen. The *network monitor* queries port statistics from physical switches using SNMP and reports current utilization for each physical link.

---

[7] Multipath routing (e.g., ECMP) based on layer 4 headers may interfere with our ability to do so; we leave this as an issue for future work.

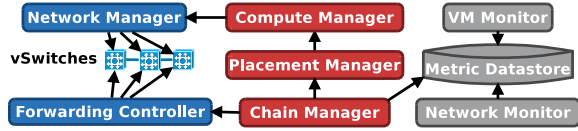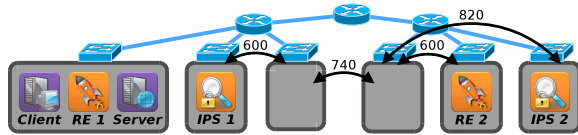**Figure 7: Stratos prototype implementation**

Figure 8: Initial instance placement and background traffic patterns for provisioning evaluation

Figure 9: Timeline of load changes and provisioning actions

7. Evaluation

We use a combination of testbed (§7.1) and simulation experiments to evaluate Stratos' ability to satisfy our key requirements (§2):

- First, we examine the ability of the Stratos forwarding plane to correctly and efficiently realize complex MBox chains in the presence of dynamics. (§7.2.)
- Second, we measure how adequately and efficiently Stratos' provisioning mechanisms satisfy application objectives, highlighting how key aspects of our provisioning process contribute to the observed outcomes. (§7.3.)
- Last, we establish the scalability of Stratos' forwarding and resource controllers. (§7.4.)

7.1 Testbed Setup

The majority of our evaluation is conducted in a small cloud testbed. The testbed consists of 36 machines (quad-core 2.4GHz, 2.67GHz, or 2.8GHz CPU and 8GB RAM) deployed uniformly across 12 racks. Each machine, running Xen and Open vSwitch, has 3 VM slots. The racks are connected in a simple tree topology with 12 top-of-rack (ToR) switches, 3 aggregation switches, and 1 core switch.

We run a variety of MBoxes, including an IPS (Suricata [13]), redundancy eliminator (SmartRE [16]), and two synthetic Click-based [32] MBoxes: *passthrough* forwards packets unmodified, and *mangler* rewrites packets' source IP (in the forward direction) and destination IP (in the reverse). We also run Apache web server and a custom workload generator. The workload generator runs 8 client threads that draw tokens from a bucket filled at a specified rate. For each token, a client thread issues an HTTP POST of a fixed size (0.2KB, 10KB, 50KB, or 100KB), and receives a reply of the same size, with a request timeout of 1 second. Client threads block if no tokens are available; the number of outstanding tokens (maximum 100) indicates unmet demand.

We generate background traffic between pairs of machines using iperf to send UDP packets at a fixed rate.
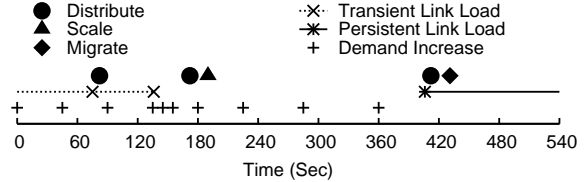
7.2 Composition Efficiency and Correctness

We first examine the ability of Stratos' forwarding controller to efficiently realize complex MBox chains in the presence of dynamics.

Efficiency. First, we measure the inflation in per-flow request completion time per-flow caused by the need to install per-flow rules. We construct a chain consisting only of a workload generator and web server (placed on different physical machines), and forward traffic between them with and without the Stratos' forwarding plane. We generate 100 requests/sec for 10 minutes and vary the request sizes for each run between 0.2KB and 100KB. Comparing the average request completion time with and without Stratos' forwarding plane, we observe that the average inflation is no more than 1ms per flow. (In general, for a chain with N mangling MBoxes, the increase in latency will be N ms higher in the worst case.)

We also benchmarked the performance of our tag-based forwarding and found that the overhead is minimal; this is consistent with prior reports on the performance of the OpenvSwitch dataplane [39].

Correctness. To evaluate affinity, we pick a chain with a workload generator, passthrough MBox, mangler MBox, and web server. Every two minutes, we add an instance of one of the MBoxes and compute a new flow distribution. We cycle through the MBoxes in round robin order. We repeat the process for two rounds. Throughout an experiment, we generate a constant workload of 50KB requests at a rate of 50 requests/sec.

We run tcpdump on each MBox during an experiment and afterwards feed each trace to Bro [38] which produces a log of all connections. For each flow (which Bro identifies with a hash based on key packet header fields), we compare the flags, byte counts, and packet counts across the connection logs for the workload generator, web server, and each MBox. Since the mangling MBox modifies packet headers, and hence results in different connection hashes, we only compare the connection logs from each half of the chain. If affinity were to be broken, discrepancies can arise in the logs; e.g., connections whose packets are split between multiple MBox instances will appear in the logs of multiple MBoxes and the log entries will indicate that not all key packets (e.g., SYN and FIN) were seen by a given instance.

We conducted the above experiment on two chains—workload generator, passthrough, mangler, server and workload generator, passthrough, passthrough, server—and found no such discrepancies in the logs, thus affirming that Stratos' steering ensures affinity.

We note that our approach of unsharing MBoxes to handle mangling is correct by construction; nevertheless, we also validated the correctness using a similar logging approach. We do not discuss this in the interest of brevity.

7.3 Provisioning Adequacy and Efficiency

We use a single chain consisting of a workload generator, an RE MBox, an IPS MBox, and a web server. Two initial instances of each MBox are placed at fixed locations, as shown in Figure 8, to simulate placements that might occur in a cloud with existing tenants. The workload we use (the *Demand* line in Figure 10(a)) starts with 90 requests/sec and increases by 10 requests/sec at varying frequencies to reach an ending rate of 175 requests/sec after 9 minutes; the size of each request is 100KB.

Throughout the experiment, we apply three different background traffic patterns, shown in Figure 8: (*i*) at experiment start, 600Mbps of traffic is exchanged between a pair of racks under each of two aggregation switches; (*ii*) 75 seconds into the experiment, the background traffic switches to 740Mbps between a rack under each of the two aggregation switches, and this lasts for 1 minute; (*iii*) roughly 6.75 minutes into the experiment, the background traffic switches to 820Mbps between a pair of racks under one of the aggregation switches.

An illustrative run. The provisioning actions taken by Stratos, along with workload and link load changes are shown in Figure 9. Stratos leverages each of its provisioning mechanisms at appropriate points in the scenario: flow distribution occurs when transient network load shifts (and before all other provisioning action), scaling occurs following a significant increase in demand, and migration occurs when a persistent network load is introduced. We validate each of the actions below in our discussion of how well Stratos addresses application objectives and efficiency.

To understand how different pieces contribute to Stratos' performance, we compare Stratos against three alternative designs that progressively exclude specific Stratos components: (1) *HeavyWgt*, a system that does not proactively use flow distribution to alleviate bottlenecks but only invokes it after scaling or migration events; (2) *LocalView*, which is similar to HeavyWgt, except that it only monitors the CPU, memory, and network link consumption at VMs to identify bottlenecks, and initiates scaling (it does not consider network effects beyond access links of VMs, and hence it does not try migration); and (3) *UniformFlow*, which is similar to LocalView, except flows are uniformly distributed across MBox instances (as opposed to invoking Stratos' flow distribution) following horizontal scaling.

Application Objectives. While the design of Stratos is quite general and can accommodate many different types of application SLOs (i.e., in our evaluation, we consider two such metrics – throughput and request backlog – shown in Figure 10(a) to illustrate Stratos' ability to satisfy application objectives. Specifically, we configure Stratos to initiate the provisioning process (Figure 5). whenever average request latency exceeds 45ms and/or the backlog exceeds 10 requests for at least a 10 second time period.

Figure 10(a) shows that Stratos' provisioning actions restore application performance to acceptable levels relatively quickly: within 10 secs during the transient link load change (that occurs at 75 secs) and the demand spike (that occurs at 155 secs), and within ≈ 50 secs when the persistent link load change occurs (at 405 secs). The response takes longer in the latter case because migration is more heavyweight than flow distribution or scaling; it takes 45 secs to perform a VM launch (35 secs) and termination (10 secs), plus there is a 20 sec delay between flow distribution and detection of network bottlenecks while Stratos waits to see if flow distribution was sufficient to address the bottleneck. Overall, with Stratos, application requests served closely tracks application demand.

Excluding Stratos components leads to inefficiency and/or inability to meet application objectives. Consider the time period with transient network load: HeavyWgt and LocalView have clear gaps between the demand and served load, and a full backlog. (As noted in §7.1, the maximum backlog is 100 requests.) LocalView has no appropriate mechanism to address this bottleneck at all, and application performance suffers. Without flow distribution as a first-order provisioning option, HeavyWgt attempts horizontal scaling. While this succeeds eventually, request have backlogged in the interim, HeavyWgt ends up using more instances than absolutely necessary, and management overhead (to launch a VM) is higher.

There is no bottleneck with UniformFlow at this point in the scenario because the starting flow distribution sends only half of the application workload over the link with the transient load, and there is sufficient capacity remaining on the link to handle this fraction of the load. For comparison, Stratos' initial flow distribution sends the entire workload across the affected link, but Stratos invokes flow distribution again when the transient network load occurs to reduce this load to half, resulting in the same situation as UniformFlow.

Now consider the time period with persistent congestion. Recall that such bottlenecks occur infrequently [19]. Stratos' backlog is better than all alternatives except HeavyWgt. In contrast with Stratos, which first attempts flow distribution for scalability reasons, HeavyWgt addresses the network bottleneck directly by invoking migration. View across the timeline, Stratos is better in all respects than HeavyWgt: it invokes strictly fewer heavy weight operations (such as VM launch), it results in lesser backlog on average, and it uses
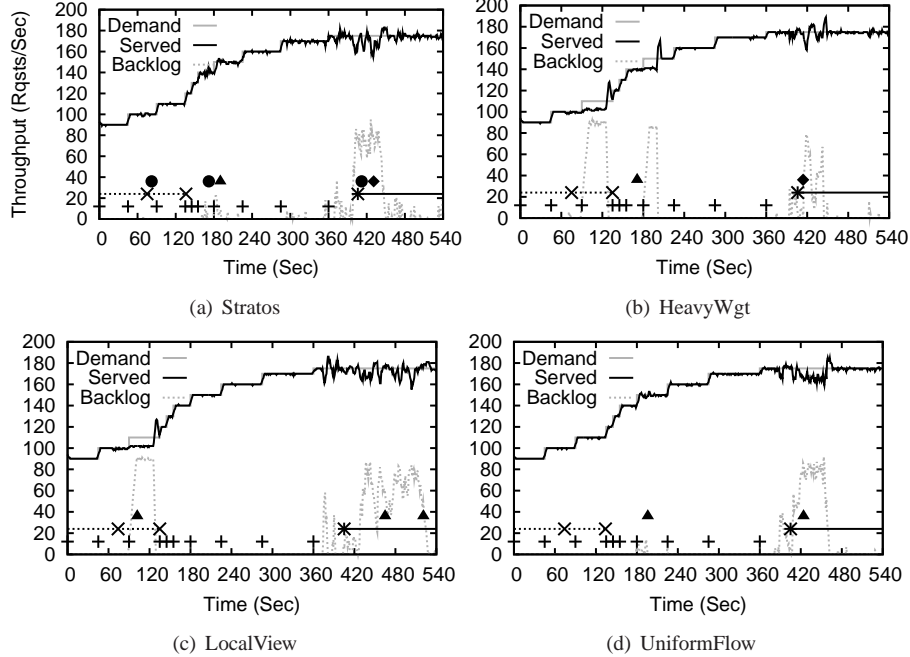
(a) Stratos

(b) HeavyWgt

(c) LocalView

(d) UniformFlow

Figure 10: Provisioning decisions and application throughput and backlog under various systems

System	# MBox Instances	Avg IPS CPU Utilization	Avg Link Utilization	
			ToR-Agg	Agg-Core
Stratos	5	70%	128Mbps	144Mbps
HeavyWgt	6	65%	128Mbps	165Mbps
LocalView	7	53%	145Mbps	146Mbps
UniformFlow	6	64%	145Mbps	188Mbps

Table 2: Efficiency under various systems

resources at least as efficiently (a topic we will cover in more detail shortly).

We found that Stratos' performance is similarly competitive with respect to application latency as well. We omit the results for brevity.

Efficiency. We examine Stratos' provisioning efficiency from three perspectives: number of MBox instances required, compute resource utilization, and network link utilization.

The number of MBox instances used to satisfy application objectives has a direct impact on the costs (for cloud infrastructure, MBox licensing, etc.) incurred by tenants. Stratos is highly effective at optimizing this metric. The RE MBox is never scaled or migrated in any of our experiments, as the packet processing capacity of a single RE instance is ≈3x higher than a single IPS instance. In contrast, the IPS MBox is horizontally scaled once with Stratos and 2-3 times with the other system designs. The extra MBox instances launched by the other systems (Table 2) hurt efficiency.

Directly related to instance count is the utilization of each instance, where higher is better as it indicates that MBox resources are being used effectively. Table 2 shows the aver-

age CPU utilization of the IPS instances. With Stratos the average utilization (70%) is 15% less than our CPU utilization threshold for compute bottlenecks (85%). In contrast, other systems have 5% to 17% lower average CPU utilization compared to Stratos.

Finally, network link utilization indicates the likelihood of future network load changes inducing bottlenecks that affect the MBox chains. Table 2 shows the average bandwidth the chain utilizes on top-of-rack switch to aggregation switch links (4 links are used) and aggregation switch to core switch links (2 links are used). We observe that the both the ToR-Agg and Agg-Core links have the lowest utilization with Stratos, while UniformFlow (the most naive approach) is the worst.

7.3.1 Provisioning at Scale

To examine Stratos's ability to efficiently satisfy application objectives across many MBox chains, we use a custom simulation framework. Our simulator places 200 chains within a 500-rack data center. The data center is arranged in a tree topology with 10 VM slots per rack and a capacity of 1Gbps on each network link. All chains have the same elements and initial instance counts: workload generators (3 instances), MBox-A (2), MBox-B (1), MBox-C (2), and servers (4); the capacity of each MBox instance is fixed at 60, 50, and 110Mbps, respectively, and the workload from each generator is 100Mbps. We perform flow distribution and horizontal scaling (no migration) for each chain until its full demand is satisfied or no further performance gain can be achieved.
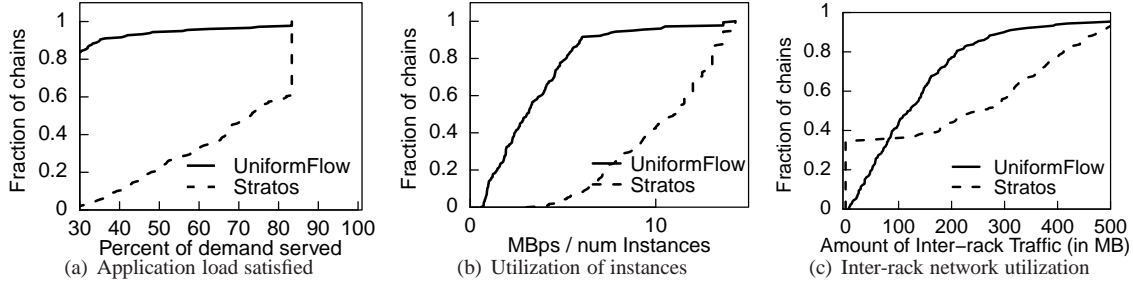
Figure 11: Simulation results to evaluate the efficiency of Stratos with a larger deployment

For ease of visualization and brevity, we only show the results comparing Stratos vs. UniformFlow, noting that the other solution strategies fall in between these two extremes.

Application Objectives. We first look at the fraction of each chain's demand that can be satisfied using Stratos and UniformFlow (Figure 11(a)). With Stratos, at least 30% of the demand for all chains is satisfied, with 85% of demand satisfied for 40% of the chains. In contrast, with UniformFlow, only 20% of chains have at least 30% of their demand satisfied.

Efficiency. Next, we examine how well the MBox instances are utilized. Figure 11(b) shows, for each chain, the volume of traffic served divided by the number of instances used. With Stratos, 90% of instances process at least 5Mbps of traffic and 50% process more than 12Mbps; with Uniform-Flow, 90% of instances process less than 5Mbps of traffic and 50% process less than 3Mbps.

Lastly, we examine the amount of inter-rack traffic generated by each chain (Figure 11(c)). Interestingly, with Stratos, a higher percent of the network is utilized by chains. This is because network-aware flow distribution allows chains to scale up more and more closely match their demand, thereby pushing more bytes out into the data center network. On the whole, the network is more effectively utilized.

7.4 Scalability of Controllers

The primary tasks of Stratos' resource controller are monitoring application performance, computing flow distributions, and placing/migrating instances. We can leverage prior techniques for scalable monitoring [29], so we focus on the resource controller's ability to perform the later two tasks for large clouds and many tenants.

We run the resource controller on a machine with an 8-core 2.27GHz CPU and 12GB of RAM. We assume a data center topology consisting of 20K racks, with 20 machines per rack, 625 aggregation switches, and 20 core switches. Inter-rack bandwidth and chain traffic volume metrics are randomly generated. We initially place 1000 tenants with 3 MBox chains each. Subsequently, we invoke either flow distribution or placement for many tenants in parallel, measuring the latency required to complete each task and the maximum sustained number of operations per second.

API	Ops/Sec	Latency	Coordinated?
Flow distribution	51	183ms	X
Placement	67	506ms	✓

Table 3: Scalability of primary provisioning tasks

Table 3 summarizes our findings. We observe that computing a flow distribution for a single tenant takes, on average, 701ms. Our prototype spends a large fraction of this time performing file I/O; solving the LP using CPLEX takes only 183ms. One controller instance can compute 51 flow distributions/sec, but we can significantly increase this capacity by running additional instances of the *chain manager* module (§6) and assigning subsets of tenants to each instance, as no synchronization between tenants is required when computing flow distributions. For placements, the sustained rate is 67 placements/sec, each of which takes 506ms on average. Placement must be coordinated among tenants, making it inherently less scalable. However, the data center can be divided into sections, with a separate *placement manager* responsible for each section. Moreover, placement (and migration) is invoked less frequently than flow distribution, requiring a lower operational capacity.

Stratos' forwarding controller can be scaled using existing approaches [33]; we exclude a scalability analysis for brevity.

8. Conclusions

Enterprises today cannot correctly and efficiently deploy virtual middleboxes to improve the performance and security of cloud-based applications. The challenges arise as a combination of two related factors: (1) the closed and proprietary nature of middlebox behaviors (e.g., resource consumption and packet modifications) and (2) the dynamic and shared nature of the cloud deployments.

To address this challenge, we designed a network-aware orchestration layer called Stratos. Stratos allows tenants to realize arbitrarily complex logical topologies by abstracting away the complexity of efficient MBox composition and provisioning. First, to ensure correct forwarding, even in the presence of middlebox mangling and dynamic provisioning, Stratos' forwarding controller combines lightweight software-defined networking mechanisms that also exploits

the virtualized nature of the deployment. Second, Stratos' provisioning controller provides a scalable network-aware strategy that synthesizes and extends techniques from traffic engineering, elastic scaling, and VM migration.

Using testbed-based live workloads and large-scale simulations, we showed that: (1) Stratos ensures efficient and correct composition; (2) Stratos' control logic can easily scale to several hundred tenants even on a single server; and (3) Stratos generates scalable yet near-optimal provisioning decisions and outperforms a range of strawman solutions illustrating that all the components in Stratos' provisioning logic contribute to the overall benefits.

References

[1] Amazon web services. `http://aws.amazon.com`.

[2] Aryaka WAN Optimization. `http://www.aryaka.com`.

[3] Cplex. `http://ibm.com/software/commerce/optimization/cplex-optimizer`.

[4] Embrane: Powering virtual network services. `http://embrane.com`.

[5] Floodlight openflow controller. `http://floodlight.openflowhub.org`.

[6] Midokura – network virtualization for public and private clouds. `http://midokura.com`.

[7] One convergence. `http://oneconvergence.com`.

[8] Open vSwitch. `http://openvswitch.org`.

[9] Plumgrid: Virtual network infrastructure. `http://plumgrid.com`.

[10] Right Scale. `http://rightscale.com`.

[11] Silverpeak wan optimization. `http://www.computerworld.com/s/article/9217298/Silver_Peak_unveils_multi_gigabit_WAN_optimization_appliance`.

[12] Snort. `http://snort.org`.

[13] Suricata. `http://openinfosecfoundation.org`.

[14] Xen. `http://xen.org`.

[15] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in Network Traffic: Findings and Implications. In *SIGMETRICS*, 2009.

[16] A. Anand, V. Sekar, and A. Akella. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In *SIGCOMM*, 2009.

[17] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

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

[19] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[20] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *SoCC*, 2011.

[21] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. Internet-Draft draft-boucadair-network-function-chaining-02, IETF Secretariat, July 2013.

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

[23] M. Dobrescu, K. Argyarki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*, 2012.

[24] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *SOSP*, 2009.

[25] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *RAID*, 2008.

[26] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions. In *HotSDN*, 2013.

[27] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-Resource Scheduling for Packet Processing. In *SIGCOMM*, 2012.

[28] V. Heorhiadi, M. K. Reiter, and V. Sekar. New Opportunities for Load Balancing in Network-Wide Intrusion Detection Systems. In *CoNEXT*, 2012.

[29] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. Star: self-tuning aggregation for scalable monitoring. In *VLDB*, 2007.

[30] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *SIGCOMM*, 2008.

[31] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *TOCS*, 18:263–297, 2000.

[33] 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.

[34] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*, 2013.

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

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

[37] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *HotCloud*, 2012.

[38] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security Symposium (SSYM)*, 1998.

[39] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *8th ACM Workshop on Hot Topics in Networks*, 2009.

[40] R. Potharaju and N. Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *IMC*, 2013.

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

[42] P. Quinn, R. Fernando, J. Guichard, S. Kumar, P. Agarwal, R. Manur, A. Chauhan, M. Smith, N. Yadav, B. McConnell, and C. Wright. Network Service Header. Internet-Draft draft-quinn-nsh-01, IETF Secretariat, July 2013.

[43] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: change you can believe in! In *HotNets*, 2011.

[44] V. Sekar, R. Krishnaswamy, A. Gupta, and M. K. Reiter. Network-Wide Deployment of Intrusion Detection and Prevention Systems. In *CoNEXT*, 2010.

[45] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *HotNets*, 2011.

[46] 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.

[47] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers. In *INFOCOM*, 2011.

[48] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (HotICE)*, 2011.

[49] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[50] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.