Programming Slick Network Functions

Bilal Anwer
Georgia Tech
bilal@gatech.edu

Theophilus Benson
Duke University
tbenson@cs.duke.edu

Nick Feamster
Princeton University
feamster@cs.princeton.edu

Dave Levin
University of Maryland
dml@cs.umd.edu

Abstract

Current approaches to in-network traffic processing involve the deployment of monolithic middleboxes in virtual machines. These approaches make it difficult to reuse functionality across different packet processing elements and also do not use available in-network processing resources efficiently. We present *Slick*, a framework for programming network functions that allows a programmer to write a single high-level control program that specifies custom packet processing on precise subsets of traffic. The Slick runtime coordinates the placement of fine-grained packet processing elements (*e.g.*, firewalls, load balancers) and steers traffic through sequences of these element instances. A Slick program merely dictates *what* processing should be performed on specific traffic flows, without requiring the programmer to specify where in the network specific processing elements are instantiated or how traffic should be routed through them. In contrast to previous work, Slick handles both the placement of fine-grained elements and the steering of traffic through specific sequences of element instances, allowing for more efficient use of network resources than solutions that solve each problem in isolation.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks] *Network Architecture and Design*

General Terms: Algorithms; Design; Experimentation

Keywords: Software-Defined Networking (SDN); Network Functions Virtualization (NFV)

1 Introduction

Recent trends suggest that network operators seek to deploy an increasing range of network functions in the network. These functions can perform arbitrary functions on packets, including access control, intrusion detection, load balancing, caching, and transcoding. It is commonly—if not always—assumed that these functions should be deployed as monolithic middleboxes [9, 16, 21, 23, 44, 49]. Until recently, these middleboxes have been deployed as vertically integrated hardware (*e.g.*, dedicated load balancers, firewalls, and other devices), although the shift towards network functions virtualization (NFV) [10] has enabled the deployment of these middleboxes in virtual machines [33].

Current approaches to NFV make it possible to place existing middleboxes in virtual machines at various points in the network and steer traffic through those middleboxes, instantiating and decommissioning instances in response to changing traffic conditions. This approach to deploy network functions imposes severe limitations. First, it requires the wholesale deployment of an existing middlebox; they do not allow an operator to implement custom, fine-grained packet processing functions in the data-plane that could be re-used across multiple applications. For example, many middleboxes may (re)implement their own packet processing modules that filter or load-balance traffic, compute statistics on traffic flows, or otherwise perform operations on packets (*e.g.*, checksums) that could be shared across different functions. Second, deploying an entire middlebox inside a virtual machine does not scale to a large number of instances on any physical machine, and deploying (or migrating) the middlebox functions may be cumbersome in their own right.

We offer a fundamentally different approach to deploy network functions. Rather than the conventional approach of redirecting traffic flows through monolithic middleboxes, we propose a programming model that allows a programmer to specify which sequences of network functions should be applied to traffic that passes through the network, leaving the thornier questions of *where* in the network those functions are actually applied and *how* these functions are applied to the underlying runtime system.

This paper presents *Slick*, an approach to programming network functions that allows an operator to implement network functions as chains of lightweight functions that can be placed at arbitrary locations in the network and composed into more complex packet processing sequences. Slick has two salient features:

- **Programming abstraction.** We develop a programming abstraction that allows a network operator to (1) implement custom network functions in a high-level language (*i.e.*, Python) and (2) specify which traffic flows should be routed through sequences of these functions. A programmer may implement (or reuse) specific functions as *elements* (a programming model that takes inspiration from Click [28]) and specify sequences of elements that should operate on specific portions of flowspace.
- **Runtime.** Slick's runtime scalably and efficiently implements the programming abstraction we have designed by decomposing network-wide packet processing into constituent functions and placing those functions at appropriate locations in the network. In contrast to existing approaches, which consider placement in the absence of steering [29, 41, 44], or vice versa [15, 18, 23], Slick takes a holistic approach, performing *both* placement of modular packet-processing elements and steering of traffic through those elements.

In contrast to NFV—which concerns the instantiation and management of existing monolithic middleboxes in virtual machines—Slick allows the placement of fine-grained functions, specified as *elements* that the programmer can write in a high-level programming language (*e.g.*, Python), making the placement of these functions more nimble, taking better advantage of available network resources, and

SOSR 2015, June 17–18, 2015, Santa Clara, CA. USA
Copyright 2015 ACM 978-1-4503-3451-8/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2774993.2774998

allowing potential reuse and sharing of network functions that are applied to traffic. Slick determines how many instances of each element should be instantiated and where individual elements should be placed ("placement"), as well as which traffic flows to direct through specific element instances ("steering"). Slick elements can be reconfigured at runtime after they are installed, and Slick policies can specify that placement or steering should change at runtime, in response to triggers from the network. For example, a middlebox that checks DNS requests against a blacklist could trigger all of the user's traffic to be steered through the closest deep-packet inspection element.

We develop several placement and steering algorithms and evaluate them on enterprise and data-center network topologies. Our evaluation shows that Slick's heuristics can achieve near-optimal network bandwidth utilization on many network topologies and can reduce the average link utilization compared to an approach that only uses consolidation by as much as a factor of two.

The rest of the paper is organized as follows. Section 2 compares Slick to other related work, including systems that have implemented placement and steering independently, as well as more theoretical treatments of middlebox control. Section 3 describes the Slick architecture, including its programming model and the runtime that it exposes to network operators. Sections 4 and 5 describe the implementation of Slick and our evaluation of Slick's placement and steering algorithms and its controller's performance on a variety of network topologies. Section 6 discusses possible directions for future work; we conclude in Section 7.

2 Related Work

Network functions virtualization (NFV) allows network operators to instantiate middleboxes in virtual machines and place those VMs at arbitrary locations in the network [10]; current approaches to NFV still treat middleboxes as monolithic entities, and do not explore how the constituent components of a middlebox might be decomposed into smaller modules. Other recent work has explored how monolithic middleboxes in a cellular network might be instantiated as virtual machines [21,49]. In contrast, Slick explores how an operator can implement individual functions in a high-level language and specify how those functions are chained together, while remaining agnostic to how those functions are replicated and installed across the network.

Programming model. Slick's programming model has two salient features: the decomposition of functions into modular elements and the use of triggers to redirect processing from an in-network element to the controller. Both of these features are inspired by previous work. Slick's use of the element abstraction is inspired by Click [28], which allowed programmers to write modular elements and compose them into packet processing pipelines on a single node. Slick differs from Click in that it constructs such pipelines across a network, and hence must address questions of both placement and steering. Extensible Open Middleboxes (xOMB) [3], RFC 3234 [5], and other work on modeling middleboxes [22] inspired the design and granularity of Slick element functions. Previous work has also proposed the use of triggers to allow one network element to signal to another [20, 27, 45, 47]; Slick incorporates this notion of triggers in a holistic programming model that supports more expressive triggers and perform other packet processing actions in response to the triggers. Although Slick's programming model draws inspiration from this previous work, none of these systems incorporate these mechanisms into a single coherent programming model, as Slick does. Although OpenNF [17] and Split/Merge [40] offer programming interfaces and control-plane mechanisms for helping operators migrate existing middleboxes, they do not allow operators to write network functions that operate on specific traffic flows in the data plane, nor do they provide mechanisms for placing network functions.

Programming Languages. Many programming languages for software defined networks can be used to express network policies [14,27,35,47,48]. Most of these languages (*e.g.*, Frenetic [14], Pyretic [35], Maple [48]) provide higher-level abstractions for programming OpenFlow [34] switches. Merlin [47] and Kinetic [27] provides some abstractions for handling events that middleboxes may raise (similar to Slick's ability to process triggers), but neither provides a mechanism for installing network functions onto machines that host these functions. None of this prior work focuses on decomposing the functions provided by monolithic middleboxes into finer-grained, reusable modules, or the placement or steering functions required to implement network-wide policies with these modules.

Steering. Charikar *et al.* [7] and ETTM [8] assume that network functions can be placed at all machines in the network and treat resource management purely as a steering problem. This approach simplifies resource management algorithms, since placing all functions on every node reduces resource management to a traffic steering problem. Unfortunately, as the number of middlebox functions proliferates—even Slick already supports about 15 distinct network functions—simply placing all functions on every node quickly becomes intractable.

Other recent work on steering [12, 39, 50] has assumed prespecified, fixed placement of middleboxes within a network and focused on developing an optimal steering mechanism that minimizes utilization. In contrast, Slick makes no such assumption about placement and must thus develop mechanisms for both steering and placement. Our evaluation demonstrates that control over placement significantly reduces both path length and average link utilization. pSwitching [23] and OpenPipes [18] provide mechanisms for steering traffic through middleboxes or hardware modules but do not offer a high-level programming model and do not propose specific steering mechanisms.

Placement. Stratos explores questions of middlebox placement to reduce inter-rack traffic in data centers [15] but focuses on placement of entire virtual machines and does not explore the placement of individual network functions, as in Slick. In contrast, Slick studies a different class of placement problems that arise when middleboxes are decomposed into constituent functions, each of which may have different resource utilization and effects on traffic flows. CoMB work explores whether multiple middleboxes can be consolidated on single physical machines [41] but studies consolidation at the granularity of virtual machines, as opposed to individual network functions. Our evaluation demonstrates that studying consolidation at the granularity of individual functions allows for different placement decisions (*e.g.*, placing elements that increase the amount of network traffic towards the end of a path, and vice versa), thus significantly reducing network utilization compared to CoMB. Sherry *et al.* explore placing existing network middleboxes in the cloud and routing traffic through these off-path middleboxes for processing [44]; in contrast, Slick enables on-path processing with fine-grained network functions that an operator writes in a high-level language.

Applications. The IETF service function chaining working group is actively exploring various applications of service function chaining [43], including in mobile and data-center networks. Yang *et al.* have studied how to enable certain applications by embedding net-

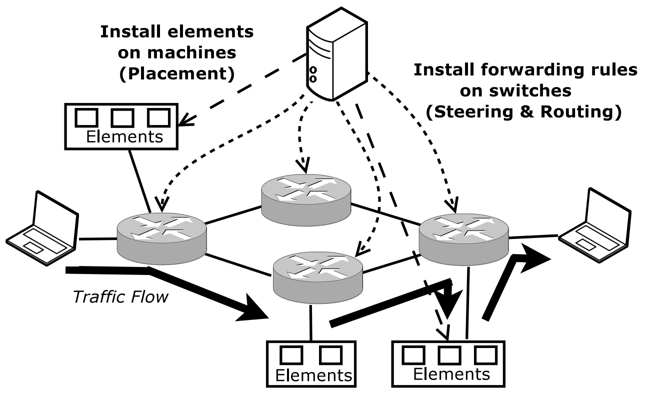

Install elements on machines (Placement)
Install forwarding rules on switches (Steering & Routing)
Elements
Traffic Flow
Elements
Elements

Figure 1: *Slick architecture. A programmer writes a Slick program that runs at the controller, which in turn installs elements (*i.e.*, high-level functions) on machines in the network (*placement*) and installs forwarding rules on switches to direct traffic flows through sequences of elements (*steering*).*

work functions in an underlying network graph, but the work focuses primarily on theoretical problems associated with embedding chains of network functions in an underlying network graph [29] and does not have a working system.

3 Slick

In this section, we present an overview of Slick, describe a motivating example (and explain why this example is difficult to implement in existing NFV architectures), and describe Slick's programming model and runtime.

3.1 Overview

Figure 1 illustrates Slick's architecture. The Slick controller runs an application that specifies a sequence of elements that should process a particular portion of flow space. An *application* specifies which traffic should flow through specific sequences of *elements*. The Slick *controller* supports these applications by deploying elements (on top of a *shim* on each machine) and installing forwarding rules in the switches to direct traffic through particular sequences of elements. The controller instantiates functions on *machines* and installs forwarding rules in *switches* to steer traffic towards those machines. The Slick runtime takes a high-level policy and determines the number of element instances to deploy (and where to deploy them) to ensure that no single element or network link is overloaded and that traffic sees good end-to-end performance. Given values for each packet-header field, the controller determines the sequence of elements that should be applied to a particular flow and installs flow table modifications into corresponding switches to ensure that the respective flow is forwarded through the corresponding sequence of element descriptors.

Motivating Example. Suppose an operator configures the network so that all Web traffic traverses an intrusion detection system (IDS) [37,46]. The application specifies that all Web traffic (*i.e.*, TCP traffic with port 80) flows through an IDS element, with all packets of a TCP connection in both directions traversing the same element. The controller deploys one or more IDS elements in the network and installs rules in the switches to direct port-80 traffic through the element. As traffic demand increases, any single IDS element may become overloaded; at this point, the Slick controller instantiates a second IDS element and splits the port-80 traffic over two IDS elements, taking care to ensure that ongoing TCP connections complete at the original IDS element and only new flows traverse the second element. If the traffic demand decreases to the levels from before the controller added additional elements and all flows through the first IDS element expire, the controller reclaims resources by removing that IDS element instance. Over time, the controller monitors the machine and network load, adjusting the traffic splitting and routing to minimize congestion. The IDS itself might inspect network traffic and perform deep-packet inspection (DPI) only when it observes DNS traffic from a device on the network to a blacklisted DNS domain.

3.2 Programming Abstractions

Each network function corresponds to a software *element*. An element may be configured either at initialization time or dynamically; it may also generate an event stream that sends events to the controller. A Slick control *application* specifies a high-level policy, indicating which traffic flows should traverse a particular sequence of elements (*e.g.*, packets with destination port 80 should traverse a firewall followed by a transcoder); an operator can write such a policy independently of the network topology or where the elements are installed.

Slick supports modular, composable elements that permit reuse across many applications; each element also supports dynamic configuration and supports sending events to the Slick controller that might subsequently affect its operation. Slick elements are inspired by elements in Click modular routers [28], from which we derive Slick's name. In this section, we describe how to program functions and applications, detailing the interfaces they expose and the abstractions presented to them.

3.2.1 Writing Slick Elements

Slick elements run on machines; an element can be an arbitrary executable and may also have state. Elements process packets, handle configuration requests from applications, and send events to the controller.

Element methods. When a controller first installs an element on a machine, it invokes the element's `init()` method. As packets destined for that element arrive at the machine, the element's `process_pkt()` method is called; this method can perform arbitrary packet processing. An element can also be configured dynamically by the controller: the `configure()` method allows the controller to dynamically reconfigure network elements. This method also allows a controller to update an element's internal state; for example, a firewall element could accept new rules via `configure()`. Finally, an element can issue asynchronous, distributed *triggers* that allows it to send events to the controller. The `raise_trigger` method accepts arbitrary inputs and delivers them to the controller (who, as we will see, delivers them to the proper applications' trigger handlers). Figure 2 shows an example of a simple Slick element that logs all packets that it sees. The `init` method (lines 6–10) performs any operations that should be called when the element is initialized (in this case, opening a file); the `process_pkt` method (lines 12–16) is invoked whenever the element sees a packet.

Two properties of the `Element` class design make it easy to reuse elements across applications. First, elements need not specify the traffic flows that they process; an element simply processes *any* that is passed to it. Second, elements are agnostic about what application is invoking them. For example, the `TriggerAll` element sends an event to the controller, and *any* control application that registers for these events will receive them. Because any element implementation is agnostic about both the subset of traffic that it will operate

```
class Logger(Element):
    def __init__(self, shim, ed):
        Element.__init__(self, shim, ed )
        self.file_handle = None

    def init(self, params):
        filename = params["file_name"]
        filename += str(self.ed)
        if(filename):
            self.file_handle = open(filename, 'a+', 0)

    def process_pkt(self, packets):
        for buf in packets:
            flow = self.extract_flow(buf)
            self.file_handle.write(str(flow) + '\n')
        return packets
```

Figure 2: *The* Logger *element logs all packets it receives. (We have elided the element's* shutdown *method for clarity.)*

on and the applications that will instantiate it, any given element implementation can be reused across a wide variety of applications.

3.2.2 Programming Slick Applications

Slick applications run at the controller. These control applications specify a sequence of elements that should process a given portion of flow space (*e.g.*, send all port 53 traffic through a Logger element).

Instantiating elements. An application specifies a portion of flow space and applies that flow to a particular element/elements using the apply_elem() method. Applying an element to a portion of flow space causes the controller to install that element at the appropriate locations in the network.

Figure 3a shows an example HttpLogger control application. Lines 7–9 specify that the controller should ensure that Logger (Figure 2) operates on all traffic with destination port 80, and to supply http.log as its input parameter (which will set the log's filename). The apply_elem() method (Line 10) takes as inputs the flow to which an element should be applied, the name of the element, and an optional set of parameters to send to those elements' init() method. Each call to apply_elem creates a new instance of the specified elements.

A Slick application may create multiple instances of multiple elements. For example, the HttpLogger application could have made another call to apply_elem on all port 443 traffic with another Logger function to also log HTTPS traffic. The apply_elem method returns a unique element descriptor for each instantiated element, to allow the controller to configure these elements after installation time, and to process triggers.

Interacting with elements. An application can also interact with any installed element after the element has been installed in the network. Applications use configure() with the corresponding element descriptor to send arbitrary configuration messages to Slick controller, which will ultimately result in a call to that element instance's configure(). When an element sends a trigger to the controller, the controller calls the corresponding application's handle_trigger() method and passes it two values:the descriptor of the element that raised the trigger and any associated data. HttpLoggerViaTrigger in Figure 3b applies the TriggerAll element (which simply raises

```
1  class HttpLogger(Application):
2      def __init__(self, controller, ad):
3          Application.__init__(self, controller, ad)
4
5      def init(self):
6          parameters = [{"file_name":"/tmp/http_log_mach"}]
7          flow = self.make_wildcard_flow()
8          flow['tp_dst'] = 80
9          flow['nw_proto'] = 6
10         ed = self.apply_elem(flow, ["Logger"], parameters)
11         if(self.check_elems_installed(ed)):
12             self.installed = True
```

(a) *Logging all port-80 traffic at in-network traffic elements.*

```
class HttpLoggerViaTrigger(Application):
    def __init__(self, controller, ad):
        Application.__init__(self, controller, ad)

    def init(self):
        flow = self.make_wildcard_flow()
        flow['tp_dst'] = 80
        flow['nw_proto'] = 6
        self.ed = self.apply_elem(flow, ["TriggerAll"])
        if(self.check_elems_installed(self.ed)):
            self.installed = True
            self.file_handle=open("http.log", 'a')

    def handle_trigger(self, ed, msg):
        if(ed in self.ed):
            self.file_handle.write(str(msg))
```

(b) *Logging all port 80 traffic at the controller.*

Figure 3: *Two implementations of* HttpLogger *that perform logging in different locations.*

a trigger for every packet) to all HTTP traffic; handle_trigger() will thus be called with each HTTP packet sent in the network.

Choosing where functions are performed. Figures 3a and 3b illustrate how Slick's programming model allows a programmer to chose where processing takes place: (1) HttpLogger places all the work in the in-network machine by having the Logger element capture and log all of the packets to file; (2) HttpLoggerViaTrigger uses the TriggerAll element to cause the controller to log all packets at the controller application. Each of these implementations represents two *extreme* design points. The first approach places all processing at the elements themselves, which is similar to how middleboxes operate today. This approach scales well, depending on where elements are installed in the network. The latter approach places all processing at the controller, which can introduce a bottleneck at the controller.

Building applications from multiple elements. Slick applications can also define interactions between multiple elements. Figure 4 shows a BlacklistDropper application, which also illustrates the use of raise_trigger and configure in the DNSBlacklist element. The application applies DNSBlacklist element to all outgoing DNS traffic (line 5), which raises a trigger whenever it detects a DNS lookup to a blacklisted domain (lines 23–28). When the application receives this trigger, it installs the DropAll element that simply

```
1  class BlacklistDropper(Application):
2      def init(self, blacklist):
3          flow = self.make_wildcard_flow()
4          flow['tp_dst'] = 53
5          eds = self.apply_elem(flow, ["DnsDpi"])
6          if(self.check_elems_installed(eds)):
7              self.installed = True
8          droppers = list()
9
10     def handle_trigger(self, ed, trigger):
11         if(trigger['type'] == 'BlacklistedQuery'):
12            src_flow = self.make_wildcard_flow()
13            src_flow['nw_src'] = trigger['src_ip']
14            eds = apply_elem(src_flow, ["DropAll"])
15            if(self.check_elems_installed(eds)):
16                droppers.append(eds[0])
17
18
19 class DNSBlacklist(Element):
20     def init(self, blacklist):
21         self.blacklist = blacklist
22
23     def process_pkt(self, pkts):
24         domain, src_ip = extract_dns_domain(pkts)
25         if(domain in self.blacklist):
26             self.raise_trigger(self.ed,
27                 {'type'   : 'BlacklistedQuery',
28                  'src_ip' : src_ip })
29         return pkts
30
31     def configure(self, params):
32         if(params['command'] == 'set—blacklist'):
33             self.blacklist = params['blacklist']
```

Figure 4: *Slick applications can use triggers to asynchronously compose elements. Element descriptors disambiguate multiple instances of the same element.*

drops all packets (lines 11–14), applying it to all subsequent traffic from the host that initiated the DNS lookup (line 14).

Slick also allows element chains, enabling sequential processing of packet flows by the elements in the chain. For example, to log all the port 80 traffic and subsequently drop all the traffic, we can modify (line 10) in Figure 3a as follows:

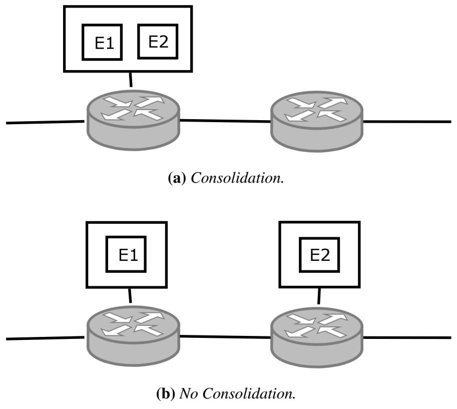
```
eds = self.apply_elem(flow, ["Logger", "DropAll"], parameters)
```

3.3 Runtime

The Slick *controller* maps a control application's high-level policy to the available pool of network resources (*i.e.*, available network bandwidth and computational elements). Given a high-level policy, the controller determines how many instances of each element to deploy and where to place or migrate them (*placement*). The controller also determines the paths that each traffic flow should take through the network so that traffic flows are processed by the correct sequence of elements and also experience good end-to-end performance (*steering*). The controller must adapt to topology changes and machine failures, as well as shifts in load and changes in the high-level policy. A *shim* on each machine allows the controller to interact with the elements (*e.g.*, to configure the element and receive triggers).

Heuristic	Reduce b/w utilization	Reduce resource utilization	Reduce latency
Consolidation	Yes	Yes	Yes
Inflation	Yes	Yes	

Table 1: *How different placement heuristics help achieve Slick objectives.*



(a) *Consolidation.*
(b) *No Consolidation.*

Figure 5: *When making placement decisions, the Slick controller must determine whether to consolidate multiple elements on a single machine or distribute those elements across multiple machines in the network or use a combination of the two.*

The Slick controller maps each new flow to elements that are installed on machines in the network and keeps an updated view of what resources are available on each machine. Instead of performing a single optimization given resources and traffic flows, the Slick controller performs a continuous *incremental optimization* that minimizes changes to the installed configuration and ongoing network traffic flows.

3.3.1 Placement

The controller's placement algorithm determines the machines in the network where element instances should be installed. The placement algorithm may ultimately place multiple instances of the same element at different places in the network, and a single machine may also host multiple elements.

Placement aims to place instances of elements at various machines in the network to ensure that flows are processed by their corresponding element sequences while using a reasonable amount of bandwidth and machine resources and ensuring a low-latency end-to-end path. Slick uses an *inflation heuristic* to reduce the overall network bandwidth required to support element sequences and a *consolidation heuristic* to reduce both the utilization on individual links and the number of overall machines required to host element instances. Table 1 summarizes how different heuristics help achieve different placement goals or have no impact on Slick's goals. Placement applies these two heuristics in order: the controller first decides whether (and how) to consolidate elements on physical machines; second, the controller determines where to place the consolidated elements.

Step 1: Consolidating elements. When we have more than one element that should operate sequentially on a certain flow space, the first step is to decide whether we should consolidate contiguous elements onto a single machine, or if we should distribute them across multiple machines. Consider the network in Figure 5, which shows two possible configurations in which a chain of two elements can be deployed.

Element	Placement
Negative Inflation	
Access Control	Source
Firewall	Source
Intrusion Prevention	Source
Deduplicator	Source
Decapsulator	Source
Compressor	Source
Positive Inflation	
Encapsulator	Destination
Decompressor	Destination
Zero Inflation	
Network Address Translation	Any
Deep Packet Inspection	Any
Intrusion Detection	Any
Stateful load balance	Any
Stateless load balance	Any
Encrypt	Any
Decrypt	Any

Table 2: *The inflation heuristic helps determine whether an element should be placed closer to the source or closer to the destination.*

We define an *inflation factor* as $\log(f_{out}/f_{in})$, where f_{in} and f_{out} are the input and output traffic volumes, respectively. The intuition for consolidation is that elements with negative inflation factor should be placed closer to sources, and elements with positive inflation factor should be placed closer to destinations. For any ordered list of elements $(E_1, \ldots, E_n)$, we can decide places to "break" the list into any number of sub-lists, where each sub-list is placed on a single machine.

We can define the inflation factor of a machine m, λ_m, as the sum of all of the inflation factors of the respective elements placed on that machine. A negative inflation factor thus means that the consolidated elements on that node decrease overall traffic, and vice versa for a positive inflation factor. Then, for a path of length l, we can define the inflation for some consolidation along that path p, λ_p as $\sum_{i=1}^{l}(i - l/2)\cdot\lambda_i$. The brute-force consolidation algorithm searches all possible consolidation combinations to minimize total inflation. Given M possible machines on which to place a sequence of E elements, the algorithm tests $\sum_{i=1}^{E}\binom{E-1}{i-1}\cdot\binom{M}{i}$ possible combinations.

Table 2 enumerates some example elements, their inflation factors, and whether they should be placed closer to source or destination. The priority of placing an element near sources or destinations can be overridden by Slick application writer.

Step 2: Placing consolidated elements. Once minimum-cost consolidation is computed, the placement algorithm uses the flow connectivity matrix for each flow space, where c_{ij} is the number of flows from i to j. The placement algorithm identifies the longest common routing path between the source(s) and destination(s). It then places consolidated elements with negative inflation factor on the node of longest common routing path that is closest to source(s), for elements with positive inflation factor, the algorithm places the consolidated element on the node of the longest common routing path that is closest to destination(s).

Elements with inflation factors near zero should be placed at machines that minimize the average path length for all source-destination pairs in the flow space, or that have the highest betweenness centrality for all source-destination pairs that exchange traffic in a given flow space. The betweenness centrality [30] of a vertex v, c_v is given by the expression:

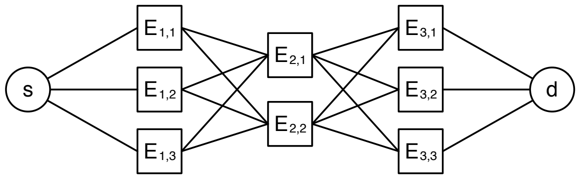


Figure 6: *Slick uses a virtual topology with m_i elements at each stage i to decide how to steer traffic from source to destination in the order specified in the Slick application.*

$$c_v = \sum_{v \notin \{s,d\}} \frac{\rho_{sd}(v)}{\rho_{sd}} \quad (1)$$

where ρ_{sd} is the total number of shortest paths between s and d and $\rho_{sd}(v)$ is the number of those paths that pass through vertex v.

3.3.2 Steering

Given elements placed in the network and a flow that must traverse a sequence of elements, the steering module determines the specific sequence of element instances that a given flow should pass through. If there are multiple instances of a particular element, the steering module determines which element instance should be used to send traffic through a particular sequence of elements. The steering module acts on a virtual topology that includes the elements and the connectivity between them.

Steering determines, for each portion of flow space, the specific sequence of element instances that should be used to process traffic for that flow. Recall that any given element might be installed in more than one place in the network; steering thus determines the instances of each element that traffic for a particular flow space should be routed through. Slick performs steering by constructing a virtual topology that represents the sources, destinations, and possible sequences of element instances at each stage of an element sequence; given this virtual topology, it computes a lowest cost path through the corresponding sequence of elements, for each portion of flow space. We describe this process in more detail below.

A Slick program determines the sequence of elements for each corresponding part of flow space; each element may have multiple instances in the network. Given an element sequence $\{E_1, \ldots, E_n\}$ for some portion of flow space, where any E_i may have multiple instances, Slick must steer each traffic flow through any instance of each element in the sequence.

To help Slick compute the appropriate sequence of element instances for each portion of flow space, we represent the set of all element instances as a virtual topology, as shown in Figure 6. Traffic from s to d is routed through one instance of E_i, in order, from E_i to E_n. Each edge in the virtual topology has a weight that corresponds to the sum of the physical network distance multiplied by the anti-log of the inflation factor. This gives us weight of each virtual edge based on the physical network topology and inflation factor of the element instances. For a flow to which n elements are to be applied, this graph takes $O(n + \prod_{i=1}^{n} m_i)$ time to construct, where m_i is the number of element instances at stage i. Given this virtual topology, Slick computes the shortest weighted path from s to d.

To avoid overloading specific element instances, Slick removes machines from the virtual topology if their load exceeds some operator-specified threshold. If no machines that host instances

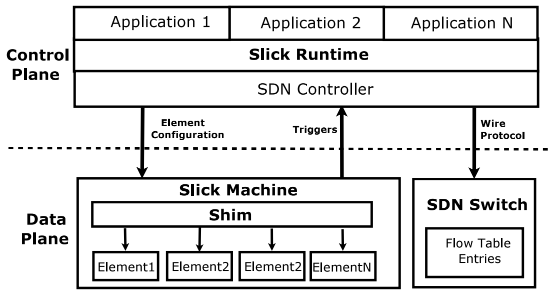


Figure 7: *The Slick runtime operates on top of an existing SDN controller (in our implementation, Pox), and hosts applications that specify functions that should operate on different parts of flow space. The controller installs and configures elements on machines in the network, which interface to the controller via a shim (*Placement*). The controller also uses a wire protocol (*e.g.*, OpenFlow) to configure flow-table entries in switches to steer traffic through the appropriate elements installed on machines (*Steering*).*

of some element E_i have spare capacity (again, determined by an operator-specified threshold), the Slick controller will provision another instance of the element on a new machine with the help of the placement module.

3.3.3 Routing

Given a specific sequence of element instances to forward traffic through, the routing module installs flow table entries into switches to ensure that a traffic flow follows a specific path between each pair of installed elements in an element sequence. It enables the steering module to implement *asymmetric steering* such that ingress and egress paths of the same flow can be asymmetric [42]. It also provides Slick runtime with network link information and placement module about the active switches generating traffic for a given flow space. Slick's routing module simply implements shortest-path routing between two element instances, although the module itself provides for other possible routing decisions between pairs of elements.

4 Implementation

We implemented Slick in about 15,000 lines of Python, with Slick's controller built on top of POX [38] controller as an SDN application. About half of the code involves the basic controller functions, such as communication with elements and interfacing to placement and steering modules, as well as the element shim as shown in Figure 7. The remainder of the code includes several elements and reference applications that use them.

4.1 Controller

The controller implementation includes functions to discover topology and machine resources, as well as the runtime that implements placement, steering and routing.

4.1.1 Discovery

The Slick controller must discover both the network topology, the machines in the network that can host packet processing elements, and the current network conditions (*e.g.*, available network resources, current machine load). It discovers topology using a link-layer discovery protocol (*e.g.*, LLDP) and machine resources through a custom resource discovery protocol.

Network topology and congestion. Network switches and servers are discovered using OpenFlow's link-layer discovery protocol (LLDP). The controller maintains a network map that includes a mapping of element instances (each of which is identified by an element descriptor) to its location in the network topology, as well as a mapping between the MAC addresses that the controller knows about and their corresponding IP addresses. The controller also periodically polls the traffic load of each network link and the amount of traffic that each element is processing.

Machine resources. Each Slick machine runs a shim layer that registers with the Slick controller; the controller keeps track of the available machines on which it can install elements. Each machine's shim has a configuration file that contains information about that machine's available resources and any other constraints that exist; this specification ensures that the controller only installs elements on machines that have both the capability and the available resources to perform the corresponding processing (*e.g.*, a configuration might ensure that a certain encryption element is only installed on machines with the corresponding hardware acceleration for cryptographic operations). These specifications also include various other parameters including the number, types, and speeds of the processors on the machine, available storage, and the operating system type and version of the machine.

Network model and overlay network abstraction. Using information about available machines and link loads, the controller builds a network model to perform operations including (1) finding machines that can host a particular element (for placement); (2) finding machines where specific elements have been installed; (3) avoid routing new traffic flows through either congested links or loaded elements. Ultimately, the controller uses these functions to construct an overlay network for each network policy that includes the elements that are pertinent to any particular flow space.

Using knowledge of the underlying network topology and machine resources, Slick places elements and maintains an overlay network topology that abstracts the physical topology. Each policy has a corresponding overlay network topology; the steering module uses this overlay network to find, for each flow, the shortest path between the source and destination that traverses a particular sequence of elements.

4.1.2 Runtime

We implemented a variety of placement, steering, and routing algorithms in Slick. Slick implements several placement algorithms, including placing elements on *k* random machines in the network, placing nodes according to centrality, placing elements on compatible machines in a round-robin fashion, and weighting placement according to centrality on a graph with nodes weighted according to traffic load. Each placement algorithm is several hundred lines of Python. Slick implements steering according to random paths through the virtual topology (Figure 6), shortest hop count through the topology, and two different shortest paths through the virtual topology: one based purely on link weights, and another where link weights are assigned according to traffic loads. Each steering algorithm is between about 50 and 200 lines of Python. Routing is based on shortest paths in the underlying topology through the sequence of elements that steering selects; for this function, we were mainly able to rely on path setup functions in Pox, but we also implemented a mechanism to route on shortest paths through the underlying topology. Slick's routing algorithm generates microflow forwarding table entries, which creates the potential for a large number of flow-table entries. Other work has explored ways to reduce the number of flow tables installed in switches, and Slick may be able to exploit these techniques [13, 39].

4.2 Elements and Applications

Shim. The Slick shim layer makes it possible to deploy and decommission elements at runtime and also includes a virtual switch to multiplex and de-multiplex traffic through these elements. The shim also allows Slick to marshal control messages between Slick's control applications running on the controller and the elements. Control messages and triggers between applications and the controller are encapsulated in JSON and sent over TCP connections.

Reference elements and applications. To demonstrate the flexibility and generality of Slick's programming model, we have implemented nearly 15 elements, which we have incorporated into several real-world applications. The Slick elements provide functions at different granularities and levels of complexity. These function include network traffic logging, TCP Stream analysis to detect OS and browsers, DNS deep packet inspection, encryption, decryption, compression, and decompression. The applications we have implemented include a traffic quarantine application that is triggered by DNS-based traffic monitoring and an application firewall.

5 Evaluation

We evaluate Slick using Mininet emulations for a variety of traffic matrices and topologies. We address the following questions: (1) What is the performance of Slick's placement and steering algorithms? (2) How efficiently does Slick place network elements and steer traffic through these elements? (3) How close are Slick's placement and steering algorithms to the optimal solution? (4) How does Slick generalize across different types of network topologies?

5.1 Experiment Setup

We evaluate Slick using the Mininet network emulator [19]. We opted for evaluating Slick using emulation rather than simulations or testbeds because emulations allow us to evaluate Slick under a variety of network topologies and with a variety of traffic matrices while ensuring that our results faithfully replicate the dynamics of real networks. We ran the Slick controller on one virtual machine and performed network emulation using Mininet on another. Each virtual machine had eight cores assigned and both VMs were running on a server with 16 cores(Intel Xeon E5620 @ 2.40GHz) and 24 GB RAM. The Mininet emulator limits our evaluations to topologies with less than 60 switches. For all evaluations, we use the applications and elements discussed in Section 4.

Topology. To demonstrate Slick's generality, we emulate a number of network topologies representing data-centers and enterprise networks. We evaluate Slick using a Fat Tree [1] network and using a canonical tree topology that is representative of small data centers [4] and enterprise networks [26]. In each topology, we assume that a Slick machine is attached to all switches within the network, so each machine that is attached to a switch can also host Slick elements. For all the experiments we use fat-tree with 20 (K=4) switches and tree topologies with 3 tiers and 15 nodes, except where stated otherwise.

Traffic Matrices. We evaluate Slick using a combination of two types of traffic matrices. (1) *East-West* traffic, emulating machine-to-machine traffic patterns which are prevalent in modern datacenters [4], (*e.g.*, MapReduce workloads). This traffic matrix generates traffic solely between end hosts within the same network; and (2) *North-South* traffic, emulating user-to-server traffic patterns that exist in a number of networks including data centers, enterprise campus networks, and WAN. Traffic is between servers at the edge and the core-devices which act as a gateway to the Internet.

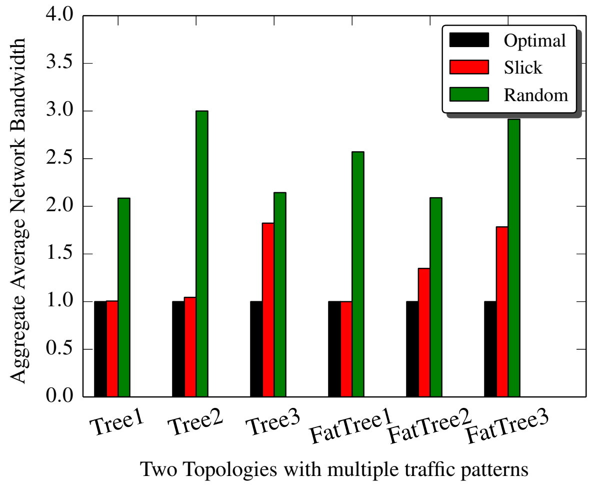


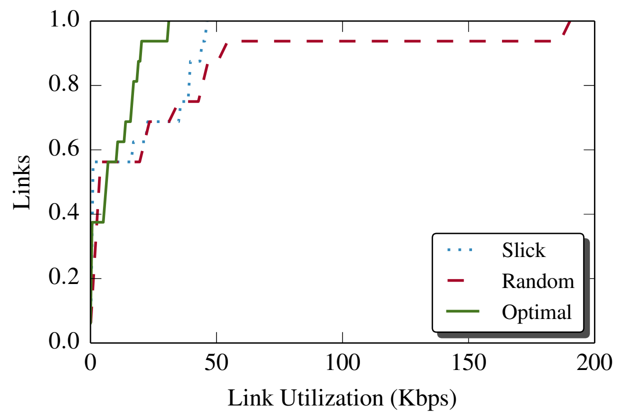
Figure 8: *Network utilization under different algorithms: Slick, Random, and Optimal.*

Evaluation Metrics. We evaluate Slick's effects on data-plane resource utilization and the performance of the Slick controller. First, we study the effects of Slick's programming model and algorithms on the overall network data plane utilization (Section 5.2). We show how using Slick's programming model and algorithms can help efficient implementation of Slick policies. To evaluate the efficiency of Slick's implementation of network policies, we focus on the following metrics: the sum of the average link utilizations (aggregate average network bandwidth), which allows us to understand the efficiency of the different algorithms; path length, which allows us to understand the impact of the different algorithms on the performance of individual flows; and link utilization, which also allows us to understand the effects of different algorithms on network traffic aggregates. In Section 5.3, we study the effects of network size, the length of Slick element chains, and the number of Slick element instances on the performance of the Slick controller.

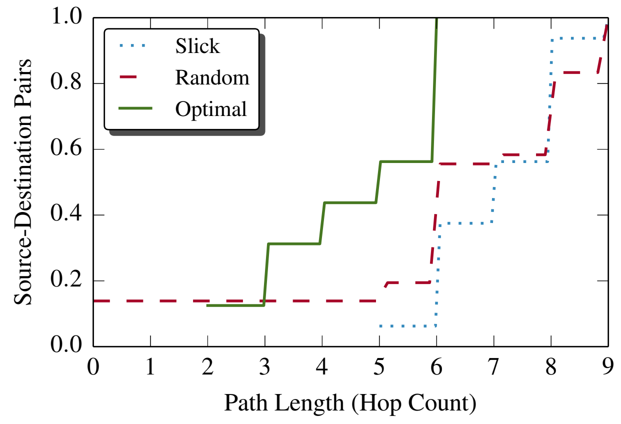
5.2 Efficiency

We now evaluate the outcomes of the placement and steering that Slick computes. In doing so, we focus on evaluating Slick's performance against an Optimal algorithm, which provides an upper bound on Slick's performance; and a Random algorithm, which provides a reasonable lower bound on Slick's performance. The Random placement algorithm randomly places elements and Random steering algorithm randomly chooses which traffic to steer through which elements, while the Optimal algorithm assumes that all elements are placed at all locations and that each node has infinite capacity, thus eliminating the need for placement and steering. The Optimal algorithm ensures that the shortest paths are used at the cost of employing more elements.

We have evaluated Slick, Random, and Optimal algorithms under all topologies and traffic matrices. Due to space constraints, we focus on the results from the largest emulations, but the results from smaller experiments are qualitatively similar. In Figure 8, we present the total bandwidth utilization from running the three algorithms on the tree topology and fat-tree topologies. The *Tree1* and *Fat-tree-1* experiments make decisions on four different flow spaces, which have both East-West and North-South traffic flows. We deploy four



(a) *Link Utilization*



(b) *Path Lengths*

Figure 9: *Comparison of Slick placement with Random and Optimal placement algorithms.*

element chains with one to two elements in each chain. For each flow space, all of the sources are clustered in single rack and all of the destinations of a flow space are in single switch rack. In the *Tree2*, *Tree3*, *Fat-Tree-2*, and *Fat-tree-3* setups, sources and destinations are randomly distributed across the network. *Tree2* and *Fat-tree-2* have eight randomly selected source destination pairs and *Tree3* and *Fat-Tree-3* have sixteen randomly selected source destination pairs.

Slick consistently outperforms Random varying between 20% and 120%. Interestingly, for the traffic matrices where sources are clustered in single rack as well as destinations, (tree-1, fat-tree1 in Figure 8), Slick performs within 5% of Optimal. For topologies where sources and destinations are randomly distributed across the network, Slick performs between Random and Optimal: consistently reducing the performance gap between Optimal and Random by half.

We also examine the link utilizations, paths lengths, and number of element instances in the resulting solutions Figure 9a and Figure 9b). Although Slick has comparable path lengths (Figure 9b) and number of elements as Random (Slick and Random use one element instance and Optimal uses 20 element instances for the Fat-Tree topology), Slick achieves much lower link utilization than Random for the same number of element instances. The link utilizations that Slick achieves are comparable to those achieved by Optimal. Moreover, Optimal can only maintain shorter paths at the cost of deploying significantly more elements: In this experiment, Optimal uses N times more elements than Slick, where N is the number of switches in the topology.

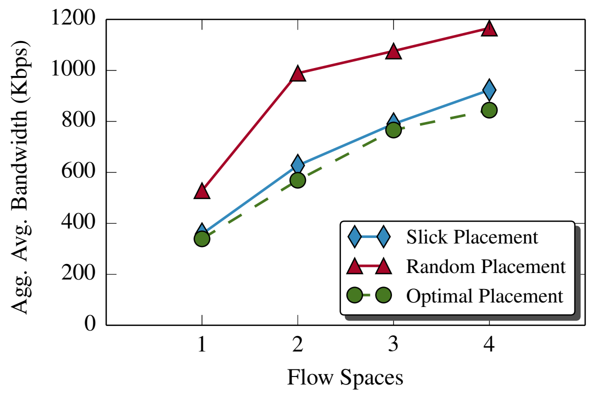


Figure 10: *Slick placement performance with increasing number of unique flow spaces that the application processes.*

Comparison to CoMB's "Strict" Consolidation. Slick uses inflation rates to guide consolidation and placement. CoMB [41]) also utilized consolidation as a way to reduce overall network utilization. CoMB argues for a strict consolidation, which *always* consolidates all elements in a chain onto one machine. We examine network utilization under CoMB and Slick's consolidation techniques and show that using inflation rates to guide consolidation can significantly reduce network utilization.

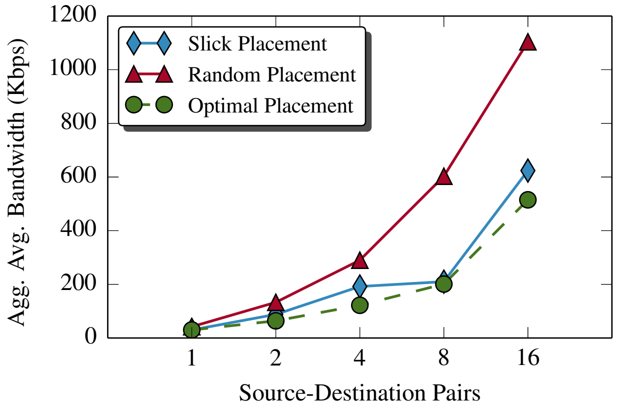
We observe that while both consolidation techniques perform comparable, there are situations when Slick consolidation outperforms CoMB's strict consolidation, reducing overall utilization by up to 50%. We examine the different element chains and observe that strict consolidation does not perform as well when chains contain a combination of elements with inflation factor > 0 and inflation factor < 0. In these cases, strict consolidation fails to account for the inflation factors and the resulting transformation in traffic that increase network utilization (*e.g.*, a decompressor/encryption element that increases the overall data transmitted).

5.2.1 General Scaling Properties

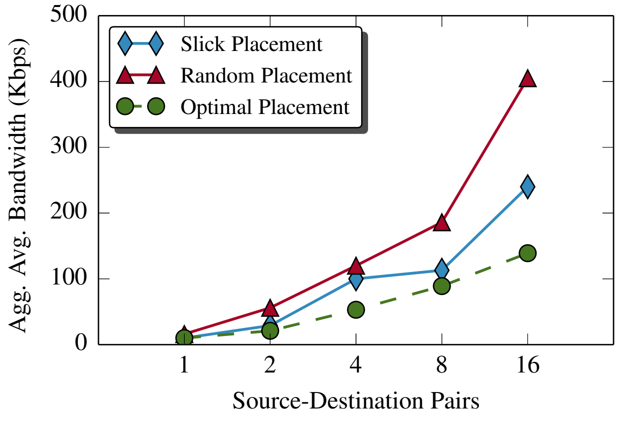
We evaluate Slick on scenarios that involve processing a different number of unique flow spaces and a random distribution of traffic sources and destinations.

Number of distinct flow spaces. Figure 10 shows the aggregate average bandwidth utilization for Slick placement and steering with varying number of flow spaces. We use a tree topology; in each run, we increase the number of flow spaces and applications and introduce an element chain in the network. We can see with increasing number of flow spaces Slick placement consistently performs within 10–15% of Optimal placement and outperforms Random placement for varying number of flow sizes. In all these experiment runs Optimal had 15 more copies of element instances than Random and Optimal, corresponding to the number of switches in the network. Each flow space had four to eight sources and four to eight destinations in it but all the sources and destinations in each flow space were non-overlapping.

Increasing Source Destination Pairs. In above experiment the intersection of sources and destinations was an empty set for all the flow spaces but both East-West and North-South traffic pattern

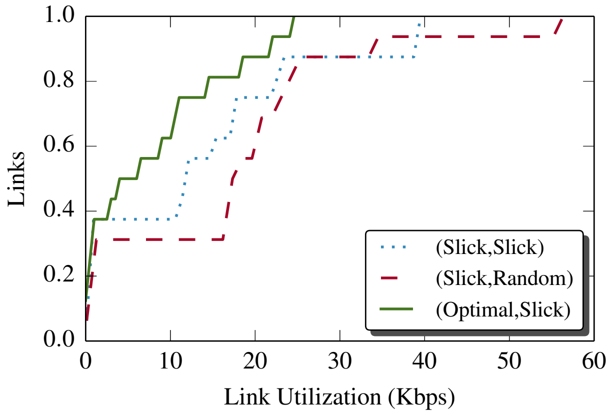


(a) *Tree Topology*

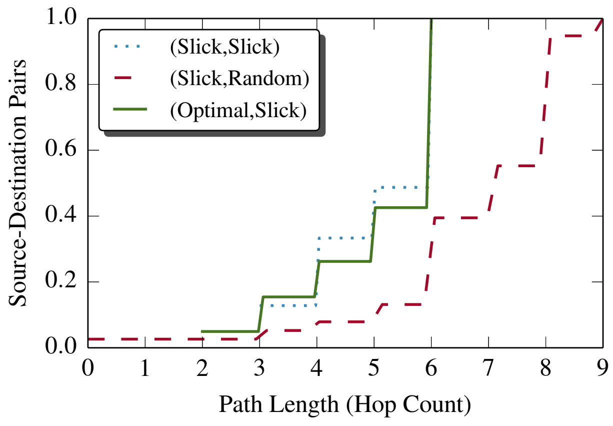


(b) *Fat-Tree Topology*

Figure 11: *Effect of different placement algorithms on traffic distribution, for different numbers of random source-destination pairs.*



(a) *Network Link Utilization.*



(b) *Path Lengths.*

Figure 12: *Quantifying the benefits of Slick's different algorithms.*

were present in them. In Figure 11, we present the aggregate average bandwidth utilization for the Slick algorithms with varying number of distributed switches all across the networks such that the intersection of source and destination switches can or cannot be an empty set. This experiment also has both North-South and East-West traffic flows. We use fixed tree and Fat-Tree topologies. Here we use a simple application with only one flow space. We deploy this application in both Tree and Fat-Tree topologies. For each experiment iteration, we randomly select source destination pairs and generate traffic between them. We increase the number of randomly selected host pairs from 1 to 16. As we can see that for both Tree 11a and Fat-Tree 11b topologies the Slick placement algorithm performance starts decreasing with increasing number of randomly distributed hosts. But for both topologies Slick placement consistently performs better than Random placement and in many cases performs comparably to Optimal.

5.2.2 Individual Placement and Steering Algorithms

We quantify the benefits of each of Slick's placement and steering algorithms by evaluating different combinations of placement and steering algorithms: (Slick,Random), a version of Slick with our placement algorithms but with Random steering; (Optimal, Slick) a version of Slick with our steering algorithm but the optimal placement; and (Slick,Slick) a version of Slick with Slick's placement and steering algorithms.

In Figure 12, we compare the link utilization and path lengths for the different algorithms. Figure 12 shows that Slick's steering algorithm contributes significantly to Slick's improvement's over random by providing reductions of both the median and 99th percentile path lengths (19%,30%) and link utilization (37%,34%) over random steering. We used a Fat-Tree topology. We deploy one element to process traffic of one flow space; with 16 randomly distributed source destination switch pairs across the network. The traffic matrix has both East-West and North-South traffic flows. Figure 12a shows that Slick's placement results in higher link utilizations in exchange for deploying fewer element instances. Figure 12b shows that Slick placement places elements in locations that provide shorter path length by restricting the number of elements used. In real-world networks, the presence of background traffic may result in higher overall traffic latencies, but we expect the results to be qualitatively similar; the respective differences in network performance between different configurations may be larger, as a reulst of this increased background traffic.

5.3 Controller Performance

We now explore the scalability of Slick's control plane and examine the different parameters that can affect its performance. We evaluate how the following dimensions: *a*) Network Size; *b*) Size of elements

Topology Size (Nodes)	Avg. Steering Time (ms)	Avg. Placement Time (ms)
15	13	131
31	15	404
63	11	581

Table 3: *Effect of network topology size on Slick's steering and placement algorithms.*

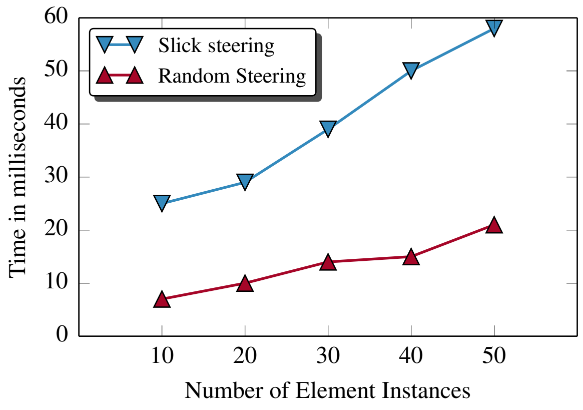


Figure 13: *Performance of Slick's steering algorithm compared to random steering.*

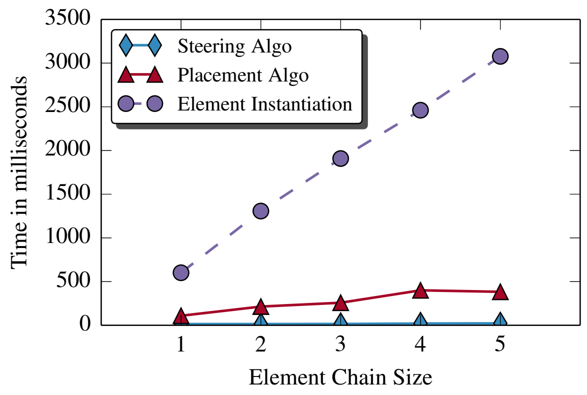


Figure 14: *Effect of element chain size on Slick algorithms.*

in a chain; *c*) Number of Element Instances in each stage of the chain. affect the run times of Slick's placement and steering modules.

Network Size and Element Chain Size. To quantify the effects of network topology and element chain size on the Slick controller's performance, we run a Slick control application with multiple flow spaces and element chains on topologies of varying sizes. From Table 3, we observe that the time for placement is linear as a function of network size. Similarly, the placement algorithm's time as well as element instantiation time is impacted by number of elements in the element chain as shown in Figure 14. Topology size and element chain size both have a profound effect on the cost of placement. We also show that steering is understandably only impacted by the number of element instances, as we explain in more detail below.

Element Instances. In this experiment, we fix the network size and size of element chain and increase the number of element instances that can potentially operate on the flow space (*i.e.*, more element instances in each stage of Figure 6). As the number of element instances in the network increases, Slick's steering algorithm's computation time increases linearly, as shown in Figure 13. Since the steering algorithm will be called only on subset of flow spaces in a network (flows requiring Slick element processing), the longer time to run the algorithm is less of a concern. Additionally, Random steering in Figure 13 shows the lower bound for computation time for any steering algorithm implemented in Slick.

6 Future Work

In this section, we discuss possible avenues for future work. We plan to release the source code to help encourage future research along the lines we have outlined in this section.

Security. The security of Slick's element placement could be improved in the following ways: (1) enforcing resource isolation between different elements on a Slick machine; (2) enforcing control over an element's ability to modify and transform packet headers and payload; and (3) default path for a new flow when elements and switches are configured. The Slick shim enforces access control and can enforce limitations on header transformations. Slick's design explicitly allows for elements to run in separate separate address spaces, thus allowing Slick to use existing primitives for resource isolation (*e.g.*, Cgroups [6, 31], Unikernels [32, 33]). Yet, more work is needed to explore exactly how various authorization policies might be expressed and implemented within the context of Slick. These questions are further complicated by the increasing prevalence of end-to-end encryption, which may require additional mechanisms for authorization and key distribution.

Element sharing. In principle, Slick makes it possible for multiple applications to share an element instance that is installed in the network. In such cases, different applications may have unique or conflicting configurations for the same element instance: For example, two different applications may use an encryption instance with different encryption keys or a compression element with different levels of compression. To support this level of flexibility, the Slick programming model would need to be extended to allow a programmer to specify which flows (and applications should map to specific element instances).

Testing and Verification. Slick includes no primitives for enabling testing, debugging, and verification of data-plane elements. Future work might explore ways to adapt existing work on automating test packet generation to test the deployment of Slick middlebox functions [24]; applying invariant checkers to determine that the modifications that Slick elements perform on packets do not result in incorrect forwarding behavior or behavior that violates other high-level policies or intent [25]; verifying that Slick policies achieve any isolation properties that a network programmer may specify [36]; and verifying that any Slick policies that are deployed do not violate end-to-end reachability properties [2, 11].

Network Routing. Slick implements several placement and steering algorithms, but only one network routing algorithm (*i.e.*, shortest path routing), yet the performance and optimality of Slick's placement and steering algorithms depend on the underlying network routing algorithm. Future work could evaluate the effectiveness of Slick's placement and steering algorithms in the context of different network routing algorithms.

7 Conclusion

Most work on managing and orchestrating middleboxes has focused on deploying monolithic middleboxes, rather than deploying individual functions written in a high-level language. Slick takes the latter approach, representing a departure from existing designs, which provides the programmer with improved flexibility and scaling properties. Slick allows a programmer to write a single application that describes a sequence of processing elements for a given part of flow space, leaving the details of how those elements are replicated and placed throughout the network to the runtime. We presented a prototype and displayed the strength of the programming model by implementing several elements and realistic applications. We showed that Slick can achieve near-optimal network bandwidth utilization for a variety of topologies and traffic loads.

Acknowledgments

We thank Jennifer Rexford for her thoughtful comments throughput the course of this project. This work was supported by DARPA through the U.S. Navy SPAWAR under contract N66001-11-C-4017 and NSF Awards CNS-1531281, CNS-1261357, and CSR-1409249.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *ACM SIGCOMM Conference*, 2008. (Cited on page 8.)

[2] E. Al-shaer, W. Marrero, A. El-atawy, and K. Elbadawi. Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security. In *IEEE ICNP*, Princeton, NJ, 2009. (Cited on page 11.)

[3] J. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers . In *Proc. ANCS*, 2012. (Cited on page 2.)

[4] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM Internet Measurement Conference*, Melbourne, Australia, Nov. 2010. (Cited on page 8.)

[5] B. Carpenter. *Middleboxes: Taxonomy and Issues*. Internet Engineering Task Force, Feb. 2002. RFC 3234. (Cited on page 2.)

[6] Control groups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. (Cited on page 11.)

[7] M. Charikar, Y. Naamad, J. Rexford, and K. Zou. Multi-Commodity Flow with In-Network Processing. Technical report, Princeton University, 2014. http://www.cs.princeton.edu/~jrex/papers/mopt14.pdf. (Cited on page 2.)

[8] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *Proc. 8th USENIX NSDI*, Boston, MA, Apr. 2011. (Cited on page 2.)

[9] Enter the Andromeda zone - Google Cloud Platform latest networking stack. http://goo.gl/u59Iw1. (Cited on page 1.)

[10] ETSI Network Function Virtualization. http://www.etsi.org/technologies-clusters/technologies/nfv. (Cited on pages 1 and 2.)

[11] S. K. Fayaz, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing Context-Dependent Policies in Stateful Data Planes. Technical Report CMU-CyLab-14-013, Carnegie Mellon University, 2014. (Cited on page 11.)

[12] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014. (Cited on page 2.)

[13] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Hong Kong, China, Aug. 2013. (Cited on page 7.)

[14] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *International Conference on Functional Programming*, Sept. 2011. (Cited on page 2.)

[15] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-Class Entities. Technical Report TR1771, University of Wisconsin-Madison, June 2012. (Cited on pages 1 and 2.)

[16] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 7–12. ACM, 2012. (Cited on page 1.)

[17] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, pages 163–174, Chicago, IL, 2014. ACM. (Cited on page 2.)

[18] G. Gibb, A. Covington, T. Yabe, and N. McKeown. OpenPipes: Prototyping high-speed networking systems, Aug. 2009. SIGCOMM 2009 Demo Session. (Cited on pages 1 and 2.)

[19] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proc. ACM SIGCOMM CoNext Conference*, 2012. (Cited on page 8.)

[20] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for Internet monitoring systems: The case for distributed triggers. In *Proc. 3nd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, San Diego, CA, Nov. 2004. (Cited on page 2.)

[21] X. Jin, E. L. Li, L. Vanbever, and J. Rexford. SoftCell: Scalable and flexible cellular core network architecture. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013. (Cited on pages 1 and 2.)

[22] D. Joseph and I. Stoica. Modeling Middleboxes. *IEEE Network*, 22(5):20–25, 2008. (Cited on page 2.)

[23] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *ACM SIGCOMM Conference*, 2008. (Cited on pages 1 and 2.)

[24] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013. (Cited on page 11.)

[25] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013. (Cited on page 11.)

[26] H. Kim, T. Benson, A. Akella, and N. Feamster. Understanding the evolution of network configuration: A tale of two campuses. In *ACM SIGCOMM Internet Measurement Conference*, Berlin, Germany, 2011. (Cited on page 8.)

[27] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. 12th USENIX NSDI*, Oakland, CA, May 2015. (Cited on page 2.)

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000. (Cited on pages 1, 2 and 3.)

[29] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. T. Wilfong, Y. R. Yang, and C. Guo. PACE: Policy-Aware Application Cloud Embedding. In *IEEE INFOCOM*, Turin, Italy, 2013. (Cited on pages 1 and 3.)

[30] F. Linton. A set of measures of centrality based upon betweenness. *Sociometry*, 40:35–41, 1977. (Cited on page 6.)

[31] Linux Containers. https://linuxcontainers.org/. (Cited on page 11.)

[32] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proc. 18th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013. (Cited on page 11.)

[33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association. (Cited on pages 1 and 11.)

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communications Review*, Apr. 2008. (Cited on page 2.)

[35] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages . In *ACM POPL*, pages 217–230, Philadelphia, USA, Jan. 2012. (Cited on page 2.)

[36] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. http://arxiv.org/abs/1409.7687, 2014. (Cited on page 11.)

[37] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999. (Cited on page 3.)

[38] POX: An OpenFlow controller. `http://www.noxrepo.org/pox/about-pox/`. (Cited on page 7.)

[39] Z. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement using SDN. In *ACM SIGCOMM Conference*, 2013. (Cited on pages 2 and 7.)

[40] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation*, Lombard, Illinois, 2013. (Cited on page 2.)

[41] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. 9th USENIX NSDI*, San Jose, CA, Apr. 2012. (Cited on pages 1, 2 and 9.)

[42] Service Function Chaining Problem Statement(IETF Draft RFC). `http://goo.gl/cQMV6k`. (Cited on page 7.)

[43] SFC: Service Function Chaining. `https://datatracker.ietf.org/wg/sfc/charter/`. (Cited on page 2.)

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone elseś problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, Helsinki, Finland, Aug. 2012. (Cited on pages 1 and 2.)

[45] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *Proc. NDSS*, 2013. (Cited on page 2.)

[46] Snort intrusion detection system. `https://www.snort.org/`. (Cited on page 3.)

[47] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G.Sirer, and N. Foster. Merlin:a language for provisioning network resources. In *Proc. CoNEXT*, Dec. 2014. (Cited on page 2.)

[48] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple:simplifying sdn programming using algorithmic policies. In *ACM SIGCOMM Conference*, 2013. (Cited on page 2.)

[49] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. ACM SIGCOMM*, Toronto, Ontario, Canada, Aug. 2011. (Cited on pages 1 and 2.)

[50] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Misra, R. Patney, E. Rubow, R. Subrahmaniam, R. Manghirmalani, M. Shirazipour, C. Truchan, and M. Tatipamula. StEERING: A Software-Defined Networking for Inline Service Chaining. In *IEEE ICNP*, Goettingen, Germany, 2013. IEEE. (Cited on page 2.)