Merlin: A Language for Managing Network Resources

Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster

***Abstract*— This paper presents Merlin, a framework for managing resources in software-defined networks. With Merlin, administrators express high-level policies using programs in a declarative language. The language includes logical predicates to identify sets of packets, regular expressions to encode forwarding paths, and arithmetic formulas to specify bandwidth constraints. The compiler maps these policies into a constraint problem that determines bandwidth allocations using parametrizable heuristics. It then generates a code that can be executed on the network elements to enforce the policies. To allow network tenants to dynamically adapt policies to their needs, Merlin provides mechanisms for delegating control of sub-policies and for verifying that modifications made to sub-policies do not violate global constraints. Experiments demonstrate the expressiveness and effectiveness of Merlin on realistic scenarios. Overall, Merlin simplifies network administration by providing high-level abstractions for specifying and enforcing network policies.**

***Index Terms*— Software-defined networking, resource management, policy delegation, formal verification.**

I. Introduction

NETWORK operators today must deal with a wide range of challenges from complex policies to a proliferation of heterogeneous devices to ever-growing traffic demands. Software-defined networking (SDN) provides tools for addressing these challenges, but existing APIs for programming SDNs are too low-level, making it difficult to effectively enforce network-wide policies. As a result, there is widespread interest in academia and industry in higher-level languages and "northbound" (i.e., application-facing) APIs that provide convenient control over network resources.

Unfortunately, current SDN languages focus mostly on packet forwarding and largely ignore issues such as bandwidth and functionality that must be implemented on middleboxes or using custom hardware [1]–[5]. Although there exist orchestration frameworks that provide mechanisms for handling a larger set of concerns including middlebox placement and traffic engineering [6]–[9], they either fail to provide a programmable API to those mechanisms, or expose APIs that are extremely simple—e.g., sequences of middleboxes. As the current interest in "intent-based" networking demonstrates [10], the challenges of managing networks using current APIs remain unmet.

This paper presents Merlin, an SDN language designed to fill this gap. Merlin provides high-level programming constructs for (i) classifying packets; (ii) controlling forwarding paths; (iii) specifying packet-processing functions; and (iv) provisioning bandwidth. These features go far beyond what can be realized using SDN switches or with existing languages like Frenetic [1], Pyretic [11], and Maple [3].

The Merlin compiler uses several advanced techniques to compute forwarding paths, map packet-processing functions to network elements, and allocate bandwidth. These techniques are based on a unified logical representation of the network that encodes both the physical topology and the policy. For traffic with bandwidth constraints, the compiler uses a mixed-integer program formulation to solve a variant of the multi-commodity flow optimization problem. For traffic without bandwidth constraints, Merlin leverages properties of regular expressions and finite automata to efficiently generate forwarding trees that respect the path constraints encoded in the logical topology. Handling these two types of traffic separately allows the compiler to provide a uniform interface to programmers while reducing the size and number of expensive constraint problems it must solve. The compiler generates configurations for a variety of network elements including switches, middleboxes, and end hosts.

Although the configurations emitted by the Merlin compiler are static, the system also incorporates mechanisms for handling dynamically changing policies. Run-time components called *negotiators* communicate among themselves to dynamically adjust bandwidth allocations and *verify* that the modifications made by other negotiators do not lead to policy violations. Again, the design of Merlin's policy language plays a crucial role. The same core constructs used by the compiler for mapping policies into a constraint problem provide a concrete basis for analyzing, processing, and verifying policies modified dynamically by negotatiors.

We have built a Merlin prototype, and used it to implement a variety of policies that illustrate the expressiveness of the language. These examples demonstrate that Merlin supports a wide range of network functionality including simple forwarding policies, policies that include bandwidth constraints, and richer packet-processing functions such as deep-packet

Manuscript received August 2, 2016; revised June 5, 2017 and December 6, 2017; accepted July 31, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Chen. This work was supported in part by Swiss NSF under Grant 200021_166132, in part by NSF under Grant CNS-1561209 and Grant CNS-1518779, Grant CNS-1111698, Grant CCF-1253165, Grant CNS-1413972, Grant CCF-1422046, Grant ACI-1440744, and Grant CCF-1535952, in part by ONR under Grant N00014-15-1-2177, in part by Cisco, in part by Facebook, in part by Fujitsu, and in part by Google. This work was done while S. Basu was at Cornell University. *(Corresponding author: Robert Soulé.)*

R. Soulé and F. Pedone are with the Faculty of Informatics, Università della Svizzera italiana, 6904 Lugano, Switzerland (e-mail: robert.soule@usi.ch).

S. Basu is with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA.

P. J. Marandi is with Microsoft Research Cambridge, Cambridge CB1 2FB, U.K.

R. Kleinberg, E. G. Sirer, and N. Foster are with the Department of Computer Science, Cornell University, Ithaca, NY 14853 USA.

Digital Object Identifier 10.1109/TNET.2018.2867239

Metavariables

Locations l	*Values* v
Network functions t	*Variables* x
Header fields f	*Natural numbers* n

Syntax

$$
\begin{array}{rrl}
\textit{Bandwidth expressions} & e & ::= n \mid x \mid e + e \\
\textit{Presburger formulas} & \phi & ::= \mathsf{max}(e, n) \mid \mathsf{min}(e, n) \\
 & & \quad \mid \phi_1 \text{ and } \phi_2 \mid \phi_1 \text{ or } \phi_2 \mid ! \ \phi_1 \\
\textit{Predicates} & p & ::= p_1 \text{ and } p_2 \mid p_1 \text{ or } p_2 \mid ! \ p_1 \\
 & & \quad \mid f = v \mid \mathsf{true} \mid \mathsf{false} \\
\textit{Path Expressions} & a & ::= . \mid l \mid t \mid a \, a \mid a | a \mid a^* \mid ! \ a \\
\textit{Statements} & s & ::= x : p \rightarrow r \\
\textit{Policies} & pol & ::= [s_1; \ldots; s_n], \phi
\end{array}
$$

Fig. 1. Merlin abstract syntax.

inspection. We have also implemented negotiators that realize max-min fair sharing and additive-increase multiplicative-decrease dynamic adaptation schemes. Our evaluation shows that the compiler can generate configurations for real-world datacenter and enterprise networks, and that Merlin can be used to obtain better application performance for data analytics and replication systems.

Overall, this paper makes the following contributions:

- It presents the design of high-level network management abstractions realized in an expressive policy language that expresses packet classification, forwarding, and bandwidth.
- It describes novel compilation algorithms that compute forwarding paths and allocate bandwidth using a mixed-integer program formulation.
- It develops techniques for dynamically adapting and verifying policies using negotiators, made possible by the language design.

This paper extends our earlier workshop [12] and conference papers [13] with expanded discussions of the optimization problem and negotiator design, additional examples, and a new experiment exploring the impact of function placement. The next section introduces Merlin's features using examples.

II. Language Design

The Merlin policy language offers constructs for specifying the behavior of the network using high-level abstractions inspired by intent-based networking [10], [14], while retaining precise control over forwarding paths and bandwidth. As an example to illustrate, suppose that we want to place a bandwidth cap on FTP traffic, while providing a bandwidth guarantee to HTTP traffic. We can encode this behavior into Merlin using several policy statements, each of which is annotated with a variable encoding the amount of bandwidth used by matching packets:

```
[ x : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dport = 20) -> .* dpi .* ;
  y : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dport = 21) -> .* ;
  z : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dport = 80) -> .* dpi *. nat .* ],
max(x + y,50MB/s) and min(z,100MB/s)
```

The statement on the first line asserts that FTP traffic (TCP port 20) from the host at IP address `192.168.1.1` to the host at address `192.168.1.2` must travel along a path that includes a packet-processing function that performs deep-packet inspection (`dpi`). The next two statements identify and constrain FTP control (TCP port 21) and HTTP (TCP port 80) traffic between the same hosts. The statement for FTP control traffic does not include any constraints on its forwarding path, while the HTTP statement includes both a deep-packet inspection (`dpi`) and a network address translation (`nat`) constraint. The formula on the final line declares a bandwidth cap (`max`) on the FTP traffic, and a bandwidth guarantee (`min`) for the HTTP traffic. Note that network functions may modify packet headers—e.g., the `nat` function rewrites IP addresses. To allow such functions to coexist with predicates on packet headers that identify sets of traffic, Merlin uses a tag-based routing scheme, as explained in Section III-D.

Formally, the syntax of the Merlin policy language is defined by the grammar in Figure 1. A policy is a set of *statements* and a *logical formula*. Each statement specifies the handling of a subset of traffic and the formula expresses a global bandwidth constraint. We require that the statements have disjoint predicates and together match all packets. In our implementation, these requirements are enforced by the compiler.

Statements: Each Merlin policy statement has several components: a *variable*, a *logical predicate*, and a *regular expression*. The variable provides a way to identify the total amount of bandwidth consumed by packets matching the predicate, while the regular expression specifies the forwarding paths and packet-processing functions that should be applied to these packets.

Logical Predicates: Merlin provides a rich predicate language for classifying packets. Atomic predicates ($f = v$) denote the set of packets whose header field f is equal to v. For instance, in the example policy above, statement `z` contains the predicate that matches packets with `ip` source address `192.168.1.1`, destination address `192.168.1.2`, and `tcp` port `80`. Merlin provides atomic predicates for standard protocols including Ethernet, IP, TCP, and UDP, and a special predicate for matching packet payloads. Predicates can be combined using conjunction (`and`), disjunction (`or`), and negation (`!`).

Regular Expressions: Merlin programmers specify forwarding paths using regular expressions. While standard regular expressions match strings of characters, Merlin regular expressions match sequences of locations (`l`) or network functions (`t`)—e.g., deep packet inspection, network address translation, content caching, proxying, traffic shaping, etc. As with POSIX regular expressions, dot (`.`) matches an arbitrary path element. The compiler determines the locations where each function can be enforced, using a mapping from function names to possible locations supplied as an input. It is free to select any matching path, provided the other constraints expressed by the policy are satisfied. Network functions must satisfy two restrictions: (i) they may only generate zero or more packets as output, and (ii) they may only access local state, which allows the compiler to freely place functions on network devices without having to worry about maintaining

global state. We assume that the set of all locations and network functions is finite.

Bandwidth Constraints: Merlin policies use logical formulas to specify constraints that either limit (`max`) or guarantee (`min`) bandwidth. In addition to conjunction (`and`), disjunction (`or`), and negation (`!`), Merlin supports an addition operator. The addition operator can be used to specify an aggregate cap on traffic, such as in the `max(x + y, 50MB/s)` term from the running example. By convention, policies without a rate clause are unconstrained. Policies that lack a minimum rate are not guaranteed any bandwidth, while policies that lack a maximum rate may send traffic at rates up to line speed. Bandwidth constraints are expressed in Presburger arithmetic—a decidable theory of first-order logic that includes addition but excludes multiplication.

Intuitively, a formula specifies the rate at which sources of various types of traffic may emit packets. Assume the universe of rates is `[0, MAX]` where `MAX` is given by physical constraints. Then `max(x, 100Mbps)` says the rate of `x` traffic must be in the interval `[0, 100Mbps)`, whereas `min(x, 100Mbps)` says the rate of `x` traffic must be in `[100Mbps, MAX]`, assuming the source is attempting to transmit that much data. Negation inverts the set of rates allowed, so that `!max(x,100Mbps)` is in fact `min(x, 100Mbps)`. Bandwidth constraints differ from packet-processing functions in one important aspect—they represent an explicit allocation of global network resources—so they require extra care during compilation.

Syntactic Sugar: Merlin supports several forms of syntactic sugar that simplify the expression of complex policies. For example, the following policy,

```
srcs := {192.168.1.1}
dsts := {192.168.1.2}
foreach (s,d) in cross(srcs,dsts):
  ip.src = s and ip.dst = d and tcp.dport = 80 ->
  .* dpi .* nat .*
```

is equivalent to statement `z` from our earlier example. The sets `srcs` and `dsts` refer to singleton sets of hosts. The `cross` operator takes the cross product of these sets. The `foreach` statement iterates over the resulting set, creating a predicate from the source `s`, destination `d`, and term `tcp.dport = 80`.

Summary: Merlin enables direct expression of high-level network policies. Programmers write policies as though they were centralized programs executing on a single device. In reality, a variety of distributed devices collaborate to collectively enforce the policy. The next few sections present Merlin's policy distribution and enforcement mechanisms in detail.

III. Compiler

The Merlin compiler performs three key tasks: (i) it translates global policies into locally-enforceable policies; (ii) it allocates bandwidth and selects forwarding paths and placement of network functions; and (iii) it generates low-level configuration instructions for devices and end hosts. To do this, the compiler takes as inputs the policy, physical topology, and a mapping from network functions to possible placements, and builds a logical topology that combines the physical topology with the constraints encoded in the policy. It then analyzes this logical topology to allocate resources and generate low-level configurations for switches, middleboxes, and end hosts.

A. Localization

Presburger arithmetic formulas are an expressive way to declare global bandwidth constraints, but implementing them leads to several challenges: aggregate guarantees can be enforced using shared quality-of-service queues on switches, but aggregate limits are more difficult, since they require distributed state in general. To solve this problem, Merlin adopts a pragmatic approach. The compiler first rewrites the formula so that bandwidth constraints apply to packets at a single location. Given a formula with one term over n identifiers, the compiler produces a new formula of n local terms that collectively imply the original. By default, the compiler divides bandwidth equally among the local terms, although other schemes are permissible. For example, the formula in our running example could be localized to:

```
max(x,25MB/s) and max(y,25MB/s) and min(z,100MB/s)
```

Rewriting policies in this way involves a tradeoff: localized enforcement increases scalability, but risks underutilizing resources. In Section IV, we describe how Merlin navigates this tradeoff via a run-time mechanism, called *negotiators*, that can dynamically adjust allocations.

B. Provisioning for Guaranteed Rates

Another challenging aspect of compilation is provisioning bandwidth. To do this, the Merlin compiler encodes the input policy and the topology into a constraint problem whose solution can be used to determine device configurations.

Logical Topology: Each policy statement contains a regular expression, which constrains the set of legal forwarding paths. To facilitate computing paths that satisfy these constraints, the compiler constructs an internal representation with a directed graph $\mathcal{G}$ in which each path corresponds to a physical path that respects the constraints expressed in a given statement. The overall graph $\mathcal{G}$ is a union of disjoint components $\mathcal{G}_i$, one for each policy statement.

Note that the regular expression a_i in statement i is over the set of locations and packet-processing functions. The first step in the construction of $\mathcal{G}_i$ is to convert a_i into a regular expression $\bar{a}_i$ over the set of locations *only*. This is done by a simple substitution: for every occurrence of a packet processor, we substitute the union of all locations associated with that function. (Recall that the compiler takes an auxiliary input specifying this mapping from functions to locations.) For example, if `h1`, `h2`, and `m1` are the three locations capable of running deep packet inspection, then the regular expression `.* dpi .*` would be transformed into `.* (h1|h2|m1) .*`. The next step is to transform the regular expression $\bar{a}_i$ into a deterministic finite automaton (DFA) $\mathcal{M}_i$ that accepts the set of strings in the regular language given by $\bar{a}_i$. This transformation is performed using the subset construction [15].

Letting L denote the set of locations in the physical network and $\mathcal{Q}_i$ denote the state set of $\mathcal{M}_i$, the vertex set of $\mathcal{G}_i$ is the Cartesian product $L \times \mathcal{Q}_i$ together with two special vertices

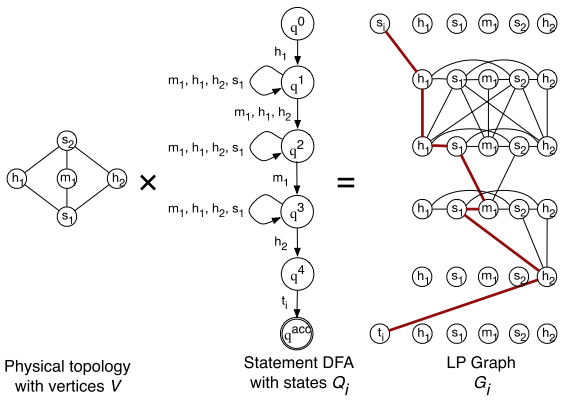


Fig. 2. Example logical topology and a possible solution.

$\{s_i, t_i\}$ that serve as a universal source and sink for paths representing statement i respectively. The graph $\mathcal{G}_i$ has an edge from (u, q) to (v, q') if and only if: (i) $u = v$ or (u, v) is an edge of the physical network, and (ii) (q, q') is a valid state transition of $\mathcal{M}_i$ when processing v. Likewise, there is an edge from s_i to (v, q') if and only if (q^0, q') is a valid state transition of $\mathcal{M}_i$ when processing v, where q^0 denotes the start state of $\mathcal{M}_i$. Finally, there is an edge from (u, q) to t_i if and only if q is an accepting state of $\mathcal{M}_i$. Paths in $\mathcal{G}_i$ correspond to paths in the physical network that satisfy the path constraints of statement i, as captured in the following lemma.

Lemma 1: A sequence of locations $u_1, u_2, \ldots, u_k$ satisfies the constraint described by regular expression $\bar{a}_i$ if and only if $\mathcal{G}_i$ contains a path of the form $s_i, (u_1, q_1), (u_2, q_2), \ldots, (u_k, q_k), t_i$ for some state sequence $q_1, \ldots, q_k$. A path in $\mathcal{G}_i$ of this form will henceforth be called a lifting of $u_1, u_2, \ldots, u_k$.

Proof: The construction of $\mathcal{G}_i$ ensures that

$$s_i, (u_1, q_1), (u_2, q_2), \ldots, (u_k, q_k), t_i$$

is a path in the graph if and only if (i) the sequence $u_1, \ldots, u_k$ represents a path in the physical network (possibly with vertices of the path repeated more than once consecutively in the sequence), and (ii) the automaton $\mathcal{M}_i$ has an accepting computation path for $u_1, \ldots, u_k$ with state sequence $q^0, q^1, \ldots, q^k$. The lemma follows from the fact that a string belongs to the regular language defined by $\bar{a}_i$ if and only if $\mathcal{M}_i$ has a computation path that accepts that string. □

Figure 2 illustrates the construction of the graph $\mathcal{G}_i$ for a statement with path expression `h1 .* dpi .* nat .* h2`, on a small example network. We assume that deep packet inspection (`dpi`) can be performed at `h1`, `h2`, or `m1`, whereas network address translation (`nat`) can only be performed at `m1`. Paths matching the regular expression can be "lifted" to paths in $\mathcal{G}_i$. The thick, red path in the figure illustrates one such lifting. Notice that the physical network also contains other paths such as `h1`, `s1`, `h2` that do not match the regular expression. These paths do not lift to any path in $\mathcal{G}_i$. For instance, focusing attention on the rows of nodes corresponding to states q^2 and q^3 of the NFA, note that all edges between these rows lead into node (m_1,q^3). Hence, any path that avoids `m1` in the physical network cannot be lifted to an s_i–t_i path in the graph $\mathcal{G}_i$.

Path Selection: Next, the compiler determines a satisfying assignment of paths that respect the bandwidth constraints encoded in the policy. The problem bears a similarity to the well-known *multi-commodity flow problem* [16], with two additional types of constraints: (i) *integrality constraints* demand that only one path may be selected for each statement, and (ii) *path constraints* are specified by regular expressions, as discussed above. To incorporate path constraints, we formulate the problem in the graph $\mathcal{G} = \bigcup_i \mathcal{G}_i$ described above, rather than in the physical network itself. Incorporating integrality constraints renders the problem NP-complete in the worst case, but several practical approaches have been developed, ranging from approximation algorithms [17]–[21], to specialized algorithms for expanders [22]–[24] and planar graphs [25], to the use of mixed-integer programming [26]. We adopt the latter technique in our implementation.

Our mixed-integer program (MIP) has a $\{0, 1\}$-valued decision variable x_e for each edge e of $\mathcal{G}$; selecting a route for each statement corresponds to selecting a path from s_i to t_i for each i and setting $x_e = 1$ on the edges of those paths, $x_e = 0$ on all other edges of $\mathcal{G}$. These variables are required to satisfy the flow conservation equations

$$\forall v \in \mathcal{G} \sum_{e \in \delta^+(v)} x_e - \sum_{e \in \delta^-(v)} x_e = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $\delta^+(v)$, $\delta^-(v)$ denote the sets of edges exiting and entering v, respectively. For bookkeeping purposes the MIP also has real-valued variables r_{uv} for each physical network link (u, v), representing what fraction of the link's capacity is reserved for statements whose assigned path traverses (u, v). Finally, there are variables $r_{\max}$ and $R_{\max}$ representing the maximum fraction of any link's capacity devoted to reserved bandwidth, and the maximum net amount of reserved bandwidth on any link, respectively. The equations and inequalities pertaining to these additional variables can be written as follows. For any statement i, let $r^i_{\min}$ denote the minimum amount of bandwidth guaranteed in the rate clause of statement i. ($r^i_{\min} = 0$ if the statement contains no bandwidth guarantee.) For any physical link (u, v), let c_{uv} denote its capacity and let $E_i(u, v)$ denote the set of all edges of the form $((u, q), (v, q'))$ or $((v, q), (u, q'))$ in $\mathcal{G}_i$.

$$\forall (u, v) \;\; r_{uv} c_{uv} = \sum_i \sum_{e \in E_i(u,v)} r^i_{\min} x_e \quad (2)$$

$$\forall (u, v) \;\; r_{\max} \geq r_{uv} \quad (3)$$

$$\forall (u, v) \;\; R_{\max} \geq r_{uv} c_{uv} \quad (4)$$

$$r_{\max} \leq 1 \quad (5)$$

Constraint 2 defines r_{uv} to be the fraction of capacity on link (u, v) reserved for bandwidth guarantees. Constraints 3 and 4 ensure that $r_{\max}$ (respectively, $R_{\max}$) is at least the maximum fraction of capacity reserved on any link (respectively, the maximum net amount of bandwidth reserved on any link). Constraint 5 ensures that the path assignment will not exceed

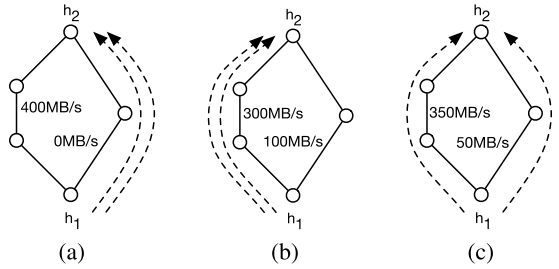


Fig. 3. Path selection heuristics. The edge labels in the graphs indicate the remaining capacities after path selection. (a) Shortest-Path (b) Min-Max Ratio (c) Min-Max Reserved.

the capacity of any link, by asserting that the fraction of reserved capacity does not exceed 1.

Path Selection Heuristics: In general, there may be multiple assignments that satisfy the path and bandwidth constraints. To indicate the preferred assignment, programmers can invoke Merlin with one of three optimization criteria:

- *Weighted shortest path:* minimizes the total number of hops in selected paths weighted by bandwidth guarantees: $\min \sum_i \sum_{u \neq v} \sum_{e \in E_i(u,v)} r^i_{\min} x_e$. This criterion is appropriate when the goal is to minimize latency.

Min-max

- *ratio:* minimizes the maximum fraction of capacity reserved on any link (i.e., $r_{\max}$). This criterion is appropriate when the goal is to balance load across links.

Min-max

- *reserved:* minimizes the maximum amount of bandwidth reserved on any link (i.e., $R_{\max}$). This criterion is appropriate when the goal is to guard against failures, since it limits the maximum amount of traffic that may be disrupted by a single link failure.

The differences between these heuristics are shown in Figure 3 which depicts a network with hosts `h1` and `h2` connected by two disjoint paths. The left path has three edges of capacity 400MB/s while the right path has two edges of capacity 100MB/s. Suppose that two statements each request 50MB/s of guaranteed bandwidth. The MIP solver will either select two-hop paths (weighted shortest path), reserve no more than 25% of capacity on any link (min-max ratio), or reserve no more than 50MB/s on any link (min-max reserved).

The Merlin compiler computes solutions that use a single path for each traffic class. While there exist approaches to multi-commodity flow that take advantage of multiple paths, we leave this extension as a topic for future work.

C. Provisioning for Best-Effort Rates

For policies that require only best-effort rates, Merlin does not need to solve a constraint problem. Instead, the compiler only needs to compute sink-trees that obey the path constraints expressed in the policy. To do this, Merlin computes the cross product of the regular expression DFA and the network topology representation, as just described, and then performs a breadth-first search over the resulting graph. To further optimize performance, the compiler uses a topology that includes only switches, and computes a sink tree for each egress switch. The compiler adds instructions to forward traffic from the egress switches to the hosts during code generation. This allows the BFS to be computed in $O(|V||E|)$, where $|V|$ is the number of switches rather than the number of hosts.

D. Code Generation

Merlin enables programmers to write high-level policies without worrying about how those policies are implemented. The Merlin compiler uses *program partitioning* to transform the policy into separate programs, instructions, and configurations that are deployed on distributed devices.

- *Switches.* Merlin generates configurations for network switches using the OpenFlow [27] libraries provided by the Frenetic SDN Controller [28]. To enforce bandwidth guarantees, Merlin uses the min-rate queues defined in version 1.0 of the OpenFlow specification, as well as device-specific port queue configuration commands.
- *Middleboxes.* For functionality such as deep packet inspection, load balancing, and intrusion detection, Merlin generates configuration scripts for Click [29] that define the sequence of packet-processing functions to apply. Other approaches are possible—e.g., Merlin could generate Puppet [30] scripts to provision and manage virtual machines instead.
- *End hosts.* Traffic filtering and rate limiting are implemented using standard Linux utilities (`iptables` and `tc`).

Tag-Based Routing: Because Merlin controls forwarding paths but also supports packet-processing functions that may modify headers (such as NAT boxes), the compiler must use a forwarding mechanism that is robust to changes in packet headers. Our implementation uses VLAN tags to encode paths to destination switches, one tag per sink tree. All packets destined for a given destination are tagged when they enter the network. Subsequent switches simply examine the tag to determine the next hop. At the egress switch, the tag is stripped off and replaced with a unique identifier for the host (e.g., the MAC address). Similar approaches are used in other systems for combining programmable switches and middleboxes such as FlowTags [31].

To sum up, the Merlin compiler is designed with flexibility in mind and can be easily extended with additional backends that capitalize on the capabilities of the various devices available in the network. Although the expressiveness of policies is bounded by the capabilities of the devices, Merlin provides a unified interface for programming them.

IV. Dynamic Adaptation

The Merlin compiler described in the preceding section translates policies into static configurations. However, these static configurations may underutilize resources, depending on how demands for traffic evolve over time. Moreover, in a shared environment, tenants may wish to customize global policies to suit their needs—e.g., adding security constraints.

To support dynamic modification of policies, Merlin uses small run-time components called *negotiators*, which transform and verify policies. Negotiators allow policy management

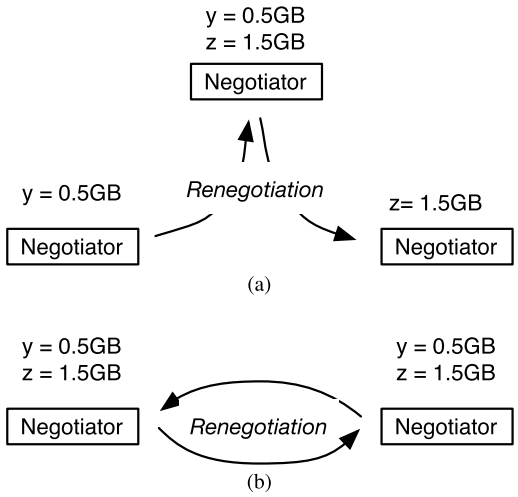


Fig. 4. Broker-based and peer-to-peer re-negotiation. (a) Broker-based. (b) Peer-to-peer.

to be delegated to tenants and provide mechanisms for verifying that choices made by tenants do not violate the original policy. Negotiators depend critically on Merlin's language-based approach—the abstractions used to express policies (i.e., predicates, regular expressions, and bandwidth constraints), also make it easy to support *verifiable* policy transformations.

Negotiators are distributed throughout the network in a tree, forming a hierarchical overlay over network elements. Each negotiator is responsible for the network elements in the subtree for which it is the root. Parent negotiators impose policies on their children. Children may refine their own policies, as long as the refinement implies the parent policy. Likewise, siblings may renegotiate resource assignments cooperatively, as long as they do not violate parent policies.

Negotiators communicate amongst themselves to dynamically adjust bandwidth allocations to fit particular deployments and traffic demands. Once agreement has been reached, the allocations can be adjusted in two ways: through a central negotiator that acts as a broker (Figure 4a) or in a peer-to-peer fashion between participant negotiators (Figure 4b).

In either case, the new allocation must conform to the parent policy. A parent policy may apply to many hosts, not just the participants in the renegotiation. At a minimum, renegotiation requires the *form* of the parent constraint, the allocations of the participants, and mappings of variables to participant hosts. Therefore, knowledge revealed during negotiation is limited to information about the participants and the global policy. Choosing between the broker and peer-to-peer strategies involves a tradeoff between performance and privacy. The broker approach is privacy-preserving, but adds the overhead of working through the broker. The peer-to-peer approach avoids the broker overhead at the expense of revealing information to end-hosts.

Merlin does not specify protocols for reaching agreement on new allocations as the details of such a protocol depend on a variety of factors, such as the trust relationships between tenants, and the tolerance for time spent reaching a consensus. These are exogenous concerns better handled outside of the core system. Our evaluation uses a peer-to-peer negotiator, and a simple protocol that assumes cooperative peers requesting reallocations in the collective best-interest.

A. Transformations

With negotiators, tenants can transform global network policies by *refining* the delegated policies to suit their demands. Tenants may modify policies in three ways: (i) the packet set specified by a statement's predicate may be further partitioned; (ii) forwarding paths may be further constrained; and (iii) bandwidth allocations may be revised.

Partitioning Packet Sets: Merlin policies classify packets into sets using predicates that combine matches on header fields using logical operators. These sets can be refined by introducing additional constraints to the original predicate. For example, a predicate for matching all TCP traffic:

```
ip.proto = tcp
```

can be partitioned into predicates that distinguish HTTP traffic:

```
ip.proto = tcp and tcp.dport = 80
ip.proto = tcp and tcp.dport != 80
```

The partitioning must be total—all packets matched by the original predicate must be matched by some new predicate.

Constraining Paths: Merlin programmers declare path constraints using regular expressions that match sequences of network locations or packet processing functions. Tenants can refine a policy by adding addition constraints to the regular expression. For example, an expression that says all packets must go through a traffic logger (`log`) function,

```
.* log .*
```

can be further constrained by adding a DPI function:

```
.* log .* dpi .*
```

A transformation that changes regular expressions is valid if the set of paths denoted by the new expression is a subset of the paths denoted by the original.

Re-Allocating Bandwidth: Merlin's limits (`max`) and guarantees (`min`) constrain allocations of network bandwidth. After a policy has been refined, these constraints can be redistributed to improve utilization. A transformation that changes bandwidth constraints is valid if the sum of the new allocations does not exceed the original allocation.

Example: As an example that illustrates the use of all three transformations, consider the following policy, which caps all traffic between two hosts at 700MB/s:

```
[x : (ip.src = 192.168.1.1 and
      ip.dst = 192.168.1.2) -> .*],
 max(x, 700MB/s)
```

This policy could be modified, as shown below. To conserve space, we have replaced the details of the policy that have not changed with an ellipsis (...).

```
[ x : (... tcp.dport = 80) -> .* log .* ;
  y : (... tcp.dport = 22) -> .* ;
  z : (... !(tcp.dport=22|tcp.dport=80)) -> .* dpi .* ],
 max(x, 500MB/s)
 and max(y, 100MB/s)
 and max(z, 100MB/s)
```

It gives 500MB/s to HTTP traffic, which must flow through a `log` box that monitors requests; it gives 100MB/s to SSH traffic, and it gives 100MB/s to the remaining traffic, which must flow through a `dpi` box.

B. Verification

In general, allowing tenants to make arbitrary modifications to policies would be unsafe. For example, a tenant could lift restrictions on forwarding paths, eliminate transformations, or allocate more bandwidth to their own traffic—all violations of the global policy set down by the administrator. Fortunately, Merlin negotiators can leverage the policy language representation to check policy inclusion, which can be used to establish the correctness of policy transformations implemented by untrusted tenants.

Intuitively, a valid refinement of a policy is one that makes it only more restrictive. To verify that a policy modified by a tenant is a valid refinement of the original, the negotiator has to check two conditions for every statement in the original policy: (i) the set of paths allowed for matching packets in the refined policy is included in the set of paths in the original, and (ii) the bandwidth constraints in the refined policy imply the bandwidth constraints in the original. These conditions can be decided using an algorithm that performs a pair-wise comparison of all statements in the original and modified policies, (i) checking for language inclusion between the regular expressions in statements with overlapping predicates [32], and (ii) checking that the sum of the bandwidth constraints in all overlapping predicates implies the original constraint.

C. Overhead

Bandwidth re-allocation does not require recompilation of the global policy, and can thus happen quite rapidly. However, changes in path constraints require global recompilation and updating forwarding rules on the switches, so they incur a greater overhead. We believe changes to paths are likely to occur less frequently than changes to bandwidth allocations.

V. EXAMPLES

To illustrate the expressiveness of the Merlin language, we present several examples inspired by realistic scenarios.

MapReduce Guarantees: MapReduce [33] uses a many-to-many communication pattern during its *shuffle* phase that results in heavy network load. Consequently, MapReduce is sensitive to background traffic in data centers, especially with protocols such as UDP that lack congestion control [34], [35]. Merlin can guarantee a minimum quality of service for MapReduce traffic:

```
[ x : (ip.src = 192.168.1.1/16 and
       ip.dst = 192.168.1.1/16 and
       ip.proto = 0x06 and
       tcp.dport = 50060) -> .* ],
  min(x,100MB/s)
```

IP Multicast Control: Monitoring applications often rely on IP multicast to transmit updates. But switches can only store a limited number of multicast addresses, and must often resort to flooding multicast packets. Instead, the administrator could maintain *multicast groups*—sets of multicast addresses with common subscribers—and compress each group to a single address [36]. Merlin can slot multicast group traffic through such a group compression function and enforce a rate limit:

```
[ x : (ip.dst = 224.0.0.1 or
       ip.dst = 224.0.0.2 )
       -> groupcompress .* ],
  max(x, 10GB/s)
```

Isolation: In some industries, regulations require corporations to keep different parts of their businesses separate. For example, the Sarbanes-Oxley Act requires the investment side of banks to be kept completely separate from the brokerage side of the company. Regulations in healthcare such as HIPAA are similar. Hence, corporate networks might need to ensure that traffic from different portions of the network never traverse the same middlebox.

```
[ x : (ip.src = 192.168.1.1/8)
       -> .* m1 .* ;
  y : (ip.src != 192.168.1.1/8)
       -> !(.* m1 .*); ]
```

Defense in Depth: A common security practice, known as defense in depth, constructs trustworthy systems by layering less trustworthy components. The following policy captures this approach by routing all traffic through two diverse firewall implementations, without enforcing a particular ordering:

```
[ x : true
      -> ( .* fire1 .* fire2 .*
         | .* fire2 .* fire1 .* ) ]
```

In summary, Merlin allows administrators to specify a diverse set of policies that can enforce bandwidth caps, provide bandwidth guarantees, indicate packet transformations, and dictate forwarding paths.

VI. IMPLEMENTATION

We have implemented a full working prototype of the Merlin system in OCaml and C. Our implementation uses the Gurobi Optimizer [37] to solve constraints, the Frenetic SDN Controller [28] to install forwarding rules on OpenFlow switches, the Click router [29] to manage software middleboxes, and the `ipfilters` and `tc` utilities on Linux end hosts. Note that the design of Merlin does not depend on these specific systems. Our implementation provides clean interfaces for incorporating different backends, allowing for others to instantiate our design with alternative systems.

Our implementation of Merlin negotiator and verification mechanisms leverages standard algorithms for transforming and analyzing predicates and regular expressions. To delegate a policy, Merlin intersects the predicates and regular expressions in each statement with those in the original policy to project out the policy for the sub-network. To verify implications between policies, Merlin uses the Z3 SMT solver [38] to check predicate disjointness, and the DPRLE library [39] to check inclusions between regular expressions.

VII. EVALUATION

To evaluate Merlin, we investigated three main issues: (i) the expressiveness of the Merlin policy language, (ii) the ability

of Merlin to improve end-to-end performance for applications, and (iii) the scalability of the compiler and negotiator components with respect to network and policy size. We used two testbeds in our evaluation. Most experiments were run on a cluster of Dell r720 PowerEdge servers with two 8-core 2.7GHz Intel Xeon processors, 32GB RAM, and four 1GB NICs. The Ring Paxos experiment (§VII-B) was conducted on a cluster of eight HP SE1102 servers equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz, with 8 GB of RAM and two 1GB NICs. Both clusters used a Pica8 Pronto 3290 switch to connect the machines. To test the scalability we ran the compiler and negotiator frameworks on various topologies and policies.

Overall, our experiments show that Merlin can effectively provision and configure real-world datacenter and enterprise networks, that it can be used to obtain better performance for big-data processing applications and replication systems, and that it enables succinctly expressing rich network policies.

A. Expressiveness

To explore the expressiveness of the Merlin policy languages, we built several network policies for the 16-switch Stanford campus backbone topology [40]. We added 24 hosts to each of the 12 edge switches in the topology and identified each pair-wise exchange of traffic between hosts as a separate traffic class. Hence, there are $(24 * 12)^2 - (24 * 12) = 82,656$ total traffic classes. We then implemented a series of policies in Merlin, and compared the sizes of the Merlin source policies and the outputs generated by the compiler. This comparison measures the degree to which Merlin is able to abstract away from hardware-level details and provide effective constructs for managing a network.

The Merlin policies we implemented are as follows:

1) *Baseline.* This policy creates pair-wise forwarding rules for all hosts in the network. The policy is restricted to only forwarding, and does not use network functions or specify bandwidth constraints. It therefore provides a baseline measurement of the number of low-level instructions that would be needed in almost any non-trivial application. The Merlin policy is only 6 lines long and compiles to 145 OpenFlow rules.
2) *Bandwidth.* This policy augments the basic connectivity by providing 10% of traffic classes a bandwidth guarantee of 1Mbps and a cap of 1Gbps. Such a guarantee could be useful, for example, to prioritize emergency messages sent to students. This policy required 11 lines of Merlin code, but generates over 1600 OpenFlow rules, 90 TC rules and 248 queue configuration commands. The number of OpenFlow rules increased dramatically due to the presence of the bandwidth guarantees which required provisioning separate forwarding paths for a large collection of traffic classes.
3) *Firewall.* This policy assumes the presence of a middlebox that filters incoming web traffic. The baseline policy is altered to forward all packets matching a particular pattern (e.g., `tcp.dport = 80`) through the middlebox. This policy requires 23 lines of Merlin code, but generates over 500 OpenFlow rules.

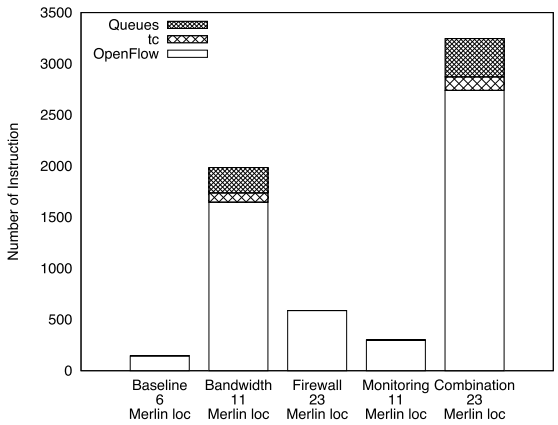


Fig. 5. Merlin expressiveness with policies for the Stanford campus topology.

4) *Monitoring.* This policy attaches middleboxes to two switches and partitions the hosts into two sets of roughly equal size. Hosts connected to switches in the same set may send traffic to each other directly, but traffic flowing between the sets must pass through a middlebox. This policy is useful for filtering traffic from untrusted sources, such as student dorms. This policy required 11 lines of Merlin code but generates 300 OpenFlow rules, roughly double the baseline.
5) *Combination*. This policy augments the baseline with a filter for web traffic, bandwidth guarantees for certain traffic classes, and an monitoring policy for certain hosts. This policy requires 23 lines of Merlin code, but generates over 3000 low-level instructions, including OpenFlow rules, TC rules, and queue configuration commands.

The results of this experiment are depicted in Figure 5. Overall, using Merlin significantly reduces the effort, in terms of lines of code, required to provision and configure network devices for a variety of real-world management tasks.

B. Application Performance

Our second set of experiments explore Merlin's ability to express policies that are beneficial for real-world applications. More specifically, they show that bandwidth provisioning and function placement improve the performance of data center applications, and also provide a proof-of-concept that Merlin can be used to effectively manage data center traffic.

Hadoop: Hadoop is a popular open-source MapReduce [33] implementation, and is widely-used for data analytics. A Hadoop computation proceeds in three stages: the system (i) applies a *map* operator to each data item to produce a large set of key-value pairs; (ii) *shuffles* all data with a given key to a single node; and (iii) applies the *reduce* operator to values with the same key. The many-to-many communication pattern used in the shuffle phase often results in heavy network load, making Hadoop jobs especially sensitive to background traffic. In practice, this background traffic can come from a variety of sources. For example, some applications, such as system monitoring tools [41], [42], network overlay management [43], and even distributed storage systems [41], [44],

use UDP-based gossip protocols to update state. A sensible network policy would be to provide guaranteed bandwidth to Hadoop and best-effort service to UDP traffic.

We implemented this policy in Merlin using just three statements. To show its impact, we ran a Hadoop job that sorts 10GB of data from a corpus of open source texts (e.g., Shakespeare's plays, etc.), and measured the time to complete it on a cluster with four servers. The cluster was configures so that all servers could act as a mapper or reducer. We ran the experiment using three different configurations:

1) *Baseline.* Hadoop had exclusive access to the network.
2) *Interference.* We simulated background traffic by using the `iperf` tool to inject UDP packets.
3) *Guarantees.* We again injected background traffic, but guaranteed 90 percent of the capacity for Hadoop.

With exclusive network access, the Hadoop job finished in 466 seconds. With background traffic causing network congestion, the job finished in 558 seconds, a roughly 20% slow down. With bandwidth guarantees, the job finished in 500 seconds, corresponding to the 90% allocation of bandwidth.

Ring-Paxos: State-machine replication (SMR) is a fundamental approach to designing fault-tolerant services [45], [46] that is used at the core of many current systems (e.g., Google's Chubby [47], Scatter [48], Spanner [49]). State machine replication provides clients with the abstraction of a highly available service by replicating the servers and regulating how commands are propagated to and executed by replicas: (i) each non-faulty replica must receive the commands in the same order; and (ii) the execution must be deterministic.

Because ordering commands in a distributed setting is non-trivial, the performance of a replicated service is often determined by the number of commands that can be ordered per time unit. To achieve high performance, state can be partitioned and each partition replicated individually (e.g., by separating data from meta-data), but the partitions will then compete for shared resources.

We assessed the performance of a key-value store service replicated with state-machine replication. Commands are ordered using an open-source implementation of Ring Paxos [50], a highly efficient implementation of the Paxos protocol [51]. We deployed two instances of the service, each one using four processes. One process in each service is co-located on the same machine and all other processes run on different machines. Clients are distributed across six different machines and submit their requests to one of the services and receive responses from the replicas.

Figure 6 (a) depicts the throughput of the two services; the aggregate throughput shows the accumulated performance of the two services. Since both services compete for resources on the common machine, each service has a similar share of the network, the bottlenecked resource at the common machine. In Figure 6 (b), we specified a guarantee of 60 percent of the capacity for the second service. Note that this guarantee does not come at the expense of utilization. If it stops sending traffic, the other service is free to use the available bandwidth.

Deep Packet Inspection: Merlin policies can also improve application performance through the careful placement of

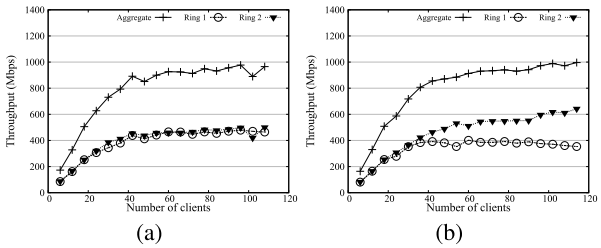


Fig. 6. Ring-Paxos (a) without and (b) with Merlin.

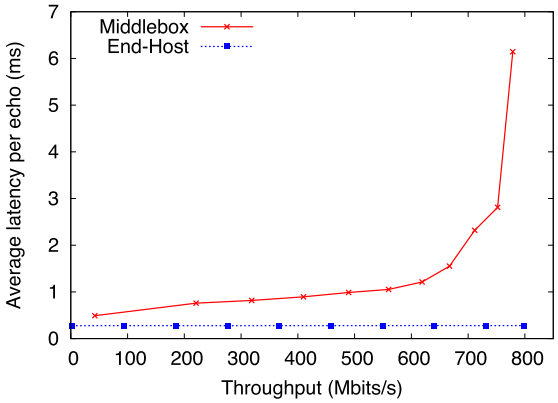


Fig. 7. A Merlin policy inspired by ETTM [52] showing that a centralized middlebox implementation of deep packet inspection has higher latency than an end-host implementation.

network functions. To demonstrate how placement can impact performance, we designed an experiment inspired by recent work on moving middlebox functionality to end-hosts [52]. We measured traffic latency for a network under two possible configurations: one in which all traffic is routed through a Deep Packet Inspection (DPI) middlebox, and one in which DPI functionality is implemented on end hosts. We implemented the DPI functionality using Click [29].

The experiment network consisted of five machines connected to a single switch. Two machines were used to generate traffic for which we measured latency. Two machines were used to generate background traffic. The final machine acted as the DPI middlebox.

We measured the latency for sending traffic under increasing network load. To place load on the network, we used the two background traffic machines. The client side forked n processes, each continually sending data to the server. We increased the amount of traffic by increasing the number of processes running concurrently. We computed the traffic throughput on these machines, and increased the load until we were unable to measure an increase in throughput. Traffic was generated using the datacenter traffic distribution identified by Greenberg *et al.* [53]. To measure latency, we used a second pair of machines. The client side sent 1000-byte probes to the server, and the server sent them back.

For each step of increasing load, we took 1000 latency measurements, and computed the 90th percentile. This eliminates extreme outliers due to packet loss and retransmission. We ran each experiment three times, and report the average results.

The results are shown in Figure 7. When the network is lightly loaded, the performance of both systems

are comparable. This is as expected, since the computation performed by end-hosts and the middlebox element are the same. The extra overhead for the middlebox case, about 0.25ms, is due to the extra hops (to and from the middlebox) that each packet needs to travel. When the network is heavily loaded, at about 800 Mbits/second, the effects of the middlebox bottleneck become manifest. The latency spikes to over 6 milliseconds, and we see an increasing number of packet drops and retransmissions. In contrast, the latency for the end-host setup stays constant at around 0.26 milliseconds. This is a 95% reduction in latency when the network is heavily loaded.

The results are exactly as one would expect, since one configuration suffers from a central bottleneck, while the other network configuration distributes the computation. However, in this case, using the shortest-path heuristic, Merlin compute an optimal placement for the network function.

Summary: Overall, these experiments show that the Merlin language can concisely express real-world policies, and that the Merlin system is able to generate code that achieves the desired outcomes for applications on real hardware.

C. Compilation and Verification

The scalability of the Merlin compiler and verification framework depends on both the size of the network topology and the number of traffic classes. Our third set of experiments evaluate the scalability of Merlin under a variety of scenarios.

Compiler: We measured the compilation time of the Merlin compiler on three different sets of network topologies.

1) *Topology Zoo.* The Internet Topology Zoo [54] dataset contains 262 topologies that represent a large diversity of network structures. We treated each node in the Topology Zoo graph as a switch, and attached one host to each switch. The topologies have an average size of 40 switches, with a standard deviation of 30 switches. We measured the compilation time needed by Merlin to determine pair-wise forwarding rules for all hosts in each topology. In other words, the policy provides basic connectivity for all hosts in the network. The results are shown in Figure 8.
2) *Balanced Trees.* We used the NetworkX Python software package [55] to generate balanced tree topologies. In a balanced tree, each node has n children, except the leaves. We treated internal node as switches, and leaf nodes as hosts. We varied the depth of the tree from 2 to 3, and the fanout (i.e., number of children) over a range of 2 to 24, to give us trees with varying numbers of hosts and switches. We identified each pair-wise exchange of traffic between hosts as a separate traffic class. We measured the compilation time for two different policies for an increasing number of traffic classes. Figure 9 (a) shows the time to provide pair-wise connectivity with no guarantees, and Figure 9 (b) shows the time to provide connectivity when 5% of the traffic classes receive bandwidth guarantees.
3) *Fat Trees.* Finally, we used the NetworkX package to generate fat tree topologies [56]. A fat tree contains a set of *pods*. Each pod of size n has two layers of $n/2$ switches. To each switch in a lower layer, we attached two hosts. Each pair-wise exchange of traffic between hosts is a separate traffic class. We increased the pod size n to create larger numbers of traffic classes. Figure 10 (a) shows the compilation time to provide pair-wise connectivity with no guarantees, and Figure 10 (b) shows the time to provide connectivity when 5% of the traffic classes receive bandwidth guarantees. To provide more detail for fat tree topologies, Figure 11 shows a sample of topology sizes and solution times for various traffic classes, along with a finer-grained accounting of compiler time.

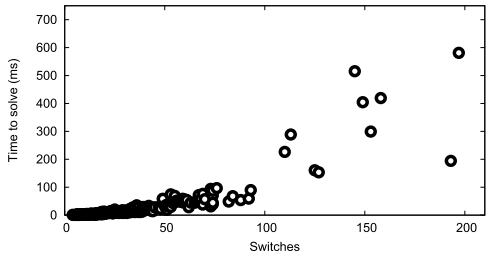
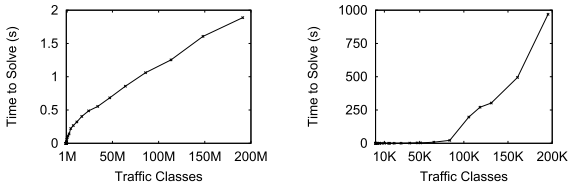
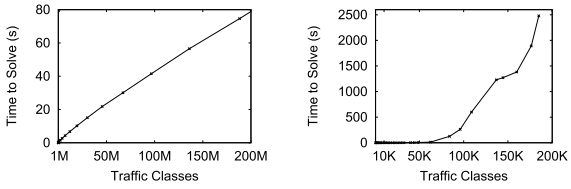


Fig. 8. Compilation times for Internet Topology Zoo.



(a) (b)

Fig. 9. Compilation times for an increasing number of traffic classes in a balanced tree topology for (a) all pairs connectivity, (b) 5% of the traffic with guaranteed priority.



(a) (b)

Fig. 10. Compilation times for an increasing number of traffic classes in a fat tree topology for (a) all pairs connectivity, (b) 5% of the traffic with guaranteed priority.

The results in Figure 8 show that for providing basic connectivity, Merlin scales well on a diverse set of topologies. The compiler finished in less than 50ms for the majority of topologies, and less than 600ms for all but one of the topologies. To improve the readability of the graph, we elided the largest topology, which has 754 switches and took Merlin 4 seconds to compile. In practice, we expect that this task would be computed offline.

Figures 9 and 10 show the impact of bandwidth guarantees on compilation time. As expected, the guarantees add significant overhead. The worst case scenario that we measured, shown in Figure 10 (b), was a network with $184,470$ total traffic classes, with $9,224$ of those classes receiving bandwidth guarantees. Merlin took around 41 minutes to find a solution.

Traffic Classes	Hosts	Switches	LP construction (ms)	LP solution (ms)	Best-Effort solution (ms)
870	30	45	25	22	33
8010	90	80	214	160	36
28730	170	125	364	252	106
39800	200	125	1465	1485	91
95790	310	180	13287	248779	222
136530	370	180	27646	1200912	215
159600	400	180	29701	1351865	212
229920	480	245	86678	10476008	451

Fig. 11. Number of traffic classes, topology sizes, and details of compilation time for fat tree topologies with 5% of the traffic classes with guaranteed bandwidth.

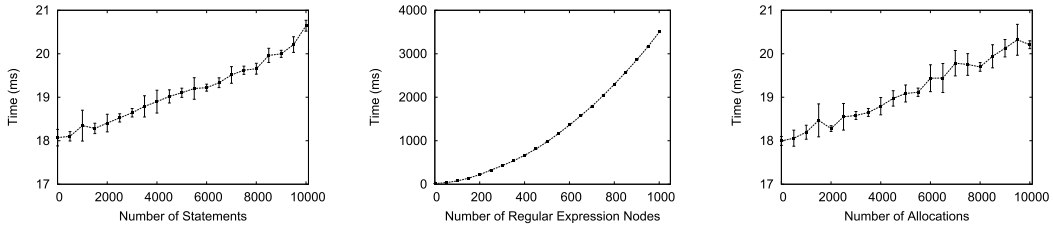


Fig. 12. Time taken to verify a delegated policy for an increasing number of delegated predicates, increasingly complex regular expressions, and an increasing number of bandwidth allocations.

Merlin finds solutions for 100 traffic classes with guarantees in a network with 125 switches in under 5 seconds.

Figure 11 shows more detail about where the compiler time is spent. The LP construction column measures how long it takes to create the LP problem. Our prototype implementation writes the problem to a file on disk before invoking the solver in a separate process. So, much of this time is attributed to string allocations and file I/O. The LP solution column measures how long it takes the solver to find a solution to the LP problem. As expected, this is where most of the time is spent as we increase the problem size. The Best-Effort solution column measures how long it takes to find paths with best-effort guarantees for the remaining traffic. The compiler spends little time finding paths that do not provide guaranteed rates.

These experiments show that Merlin can provide connectivity for large networks quickly and our mixed-integer programming approach used for guaranteeing bandwidth scales to large networks with reasonable overhead.

Verifying Negotiators: Delegated Merlin policies can be modified by negotiators in three ways: by changing the predicates, the regular expressions, or the bandwidth allocations. We ran three experiments to benchmark our negotiator verification runtime for these cases. First, we increased the number of additional predicates generated in the delegated policy. Second, we increased the complexity of the regular expressions in the delegated policy. The number of nodes in the regular expression's abstract syntax tree is used as a measure of its complexity. Finally, we increased the number of bandwidth allocations in the delegated policy. For all three experiments, we measured the time needed for negotiators to verify a delegated policy against the original policy. We report the mean and standard deviation over ten runs.

The results, shown in Figure 12, demonstrate that policy verification is extremely fast for increasing predicates and allocations. Both scale linearly up to tens of thousands of allocations and statements and complete in milliseconds. This shows that Merlin negotiators can be used to rapidly adjust to changing traffic loads. Verification of regular expressions has higher overhead. It scales quadratically, and takes about 3.5 seconds for an expression with a thousand nodes in its parse tree. However, since regular expressions denote paths through the network, it is unlikely that we will encounter regular expressions with thousands of nodes in realistic deployments. Moreover, we expect path constraints to change relatively infrequently compared to bandwidth constraints.

Dynamic Adaptation: Merlin negotiators support a wide range of resource management schemes. We implemented two common approaches: *additive-increase, multiplicative decrease* (AIMD), and *max-min fair-sharing* (MMFS). Both implementations required two components: a negotiator which ran on the same machine as the SDN controller, and end-host software, which monitors per-host bandwidth usage, and sends requests to the negotiator.

With AIMD, the end-host components send requests to the negotiator to incrementally increase their bandwidth allocation. The negotiator maintains a mapping of hosts to their current bandwidth limits. When the negotiator receives a new request, it attempts to satisfy the demand. If, however, satisfying the demand violates the global policy, it then exponentially reduces the allocation for the host. After computing the new allocations, the negotiator generates the updated Merlin policies, which are processed by the compiler to generate new `tc` commands that are installed on the end-hosts.

With MMFS, the end-host components declare resource requirements ahead of time by sending demands to the negotiator. The negotiator maintains a mapping of hosts to their demands. When the negotiator receives a new demand, it reallocates bandwidth for all hosts. It does this by attempting to satisfy all demands starting with the smallest. When there is not enough bandwidth available to satisfy any further demands, the left-over bandwidth is distributed equally among the remaining tenants. Once the new allocations are computed, the negotiator generates a new policy that reflects those allocations. The new policy is processed by the compiler to generate new queue configurations for switches, and `tc` commands for end hosts. The queue configurations ensure that satisfied

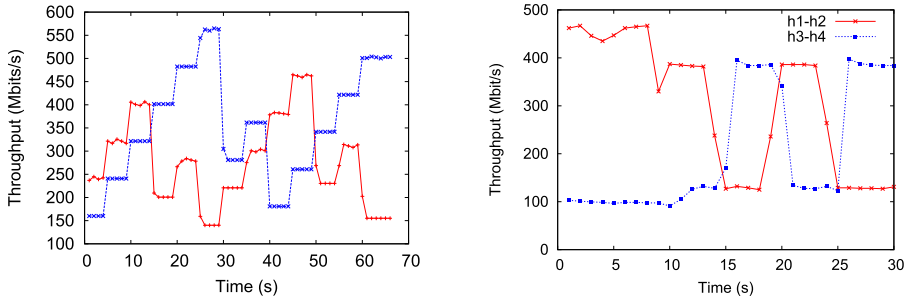


Fig. 13. (a) AIMD and (b) MMFS dynamic adaptation.

demands are respected, and the `tc` commands ensure that the remaining traffic does not exceed the allocation specified by the original policy.

Figure 13 (a) shows the bandwidth usage over time for two hosts using the AIMD strategy. Figure 13 (b) shows the bandwidth usage over time for four hosts using the MMFS negotiators. Host `h1` communicates with `h2`, and `h3` communicates with `h4`. Both expreiments were run on our hardware testbed. Overall, negotiators allow the network to quickly adapt to changing resource demands, while respecting the global constraints imposed by the policy.

VIII. Related Work

We first presented a preliminary design for Merlin in a workshop paper [12], and then described the approach in more detail, including an experimental evaluation, in a conference paper [13]. This paper build on that prior work with an expanded discussion of the optimization problem and negotiator design, additional examples, and more evaluation.

A number of systems in recent years have investigated mechanisms for providing bandwidth caps and guarantees [35], [57]–[59], implementing traffic filters [60], [61], or specifying forwarding policies at different points in the network [1], [11], [62], [63]. Merlin builds on these approaches by providing a unified interface and central point of control for switches, middleboxes, and end hosts.

SIMPLE [8] is a framework for controlling middleboxes. SIMPLE attempts to load balance the network with respect to TCAM and CPU usage. Like Merlin, it solves an optimization problem, but it does not specify the programming interface to the framework, or how policies are represented and analyzed.

The APLOMB [64] system allows network operators to specify middlebox processing services that should be applied to classes of traffic. The actual processing of packets is handled by virtual machines deployed in a cloud-based architecture. Merlin is similar, in that policies allow users to specify packet-processing functions. However, Merlin does not directly target cloud-services. Moreover, Merlin allocates paths with respect to bandwidth constraints, while APLOMB does not.

E2 [65] is a framework that implements common functionality for packet-processing applications. Similar to Merlin, it provides a declarative policy-language for traffic management. E2 differs from Merlin, though, in that it is focused on Network Function (NF) management. Users provide NF descriptions, which allow the E2 software to handle tasks such as placement, scaling, and service interconnection.

Many different programming languages have been proposed in recent years including Frenetic [1], Pyretic [2], and Maple [3]. These languages typically offer abstractions for programming OpenFlow networks. However, these languages are limited in that they do not allow programmers to specify middlebox functionality, allocate bandwidth, or delegate policies. An exception is the PANE [66] system, which allows end hosts to make explicit requests for network resources such as bandwidth. Unlike Merlin, PANE does not provide mechanisms for partitioning functionality and delegation is supported at the level of individual flows, rather than entire policies.

The Merlin compiler implements a form of program partitioning. This idea has been previously explored in a variety of other domains including secure web applications [67], and distributed computing and storage [68].

IX. Conclusion

The success of programmable network platforms has demonstrated the benefits of high-level abstractions for managing networks. Merlin further raises the level of abstraction, allowing administrators to specify the functionality of an entire network, leaving the low-level configuration of individual components to a compiler. At the same time, Merlin provides mechanisms for tailoring policies to suit specific needs, while ensuring that global constraints on forwarding paths and bandwidth are correctly enforced. This approach both simplifies network administration and lays a solid foundation for future research projects on network programmability.

References

[1] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. Int. Conf. Funct. Program.*, 2011, pp. 279–291, doi: 10.1145/2034773.2034812.

[2] C Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. Symp. Netw. Syst. Des. Implement.*, 2013, pp. 1–13.

[3] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2013, pp. 87–98 , doi: 10.1145/2486001.2486030.

[4] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," in *Proc. Symp. Princ. Program. Languages*, 2014, pp. 113–126, doi: 10.1145/2578855.2535862.

[5] T. Nelson, M. Scheer, A. D. Ferguson, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. Symp. Netw. Syst. Des. Implement.*, 2014, pp. 1–14.

[6] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proc. Workshop Hot Topics Netw.*, 2012, pp. 7–12, doi: 10.1145/2390231.2390233.

[7] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2008, pp. 51–62.

[8] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2013, pp. 27–38.

[9] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. Symp. Netw. Syst. Des. Implement.*, 2012, p. 24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228331

[10] A. Lerner, (Feb. 2017), Intent-Based Networking, Gartner Blog Network. [Online]. Available: https://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking

[11] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. Symp. Princ. Program. Lang.*, 2012, pp. 217–230, doi: 10.1145/2103656.2103685.

[12] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the Network with Merlin," in *Proc. Workshop Hot Topics Netw.*, 2013, Art. no. 24.

[13] R. Soulé *et al.*, "Merlin: A language for provisioning network resources," in *Proc. Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 213–226, doi: 10.1145/2674005.2674989.

[14] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the 'one big switch' abstraction in software-defined networks," in *Proc. Int. Conf. Emerg. Netw. Exp. Technol.*, 2013, pp. 13–24, doi: 10.1145/2535372.2535373.

[15] A. V. Aho and J. D. Ullman, *Theory of Parsing, Translation and Compiling*, vol. 1. Upper Saddle River, NJ, USA: Prentice-Hall, 1972.

[16] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, 1993.

[17] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar, "Approximation algorithms for the unsplittable flow problem," in *Proc. Int. Workshop Approximation Algorithms Combinat. Optim.*, 2002, pp. 51–66.

[18] J. Chuzhoy and S. Li, "A polylogarithmic approximation algorithm for edge-disjoint paths with congestion 2," in *Proc. IEEE Symp. Found. Comput. Sci.*, Oct. 2012, pp. 233–242.

[19] Y. Dinitz, N. Garg, and M. X. Goemans, "On the single-source unsplittable flow problem," *Combinatorica*, vol. 19, no. 1, pp. 17–41, Jan. 1999.

[20] J. M. Kleinberg, "Single-source unsplittable flow," in *Proc. IEEE Symp. Found. Comput. Sci.*, Oct. 1996, pp. 68–77.

[21] S. G. Kolliopoulos and C. Stein, "Approximation algorithms for single-source unsplittable flow," *SIAM J. Comput.*, vol. 31, no. 3, pp. 919–946, 2001.

[22] A. Z. Broder, A. M. Frieze, and E. Upfal, "Static and dynamic path selection on expander graphs: A random walk approach," *Random Struct. Algorithms*, vol. 14, no. 1, pp. 87–109, 1999.

[23] A. M. Frieze, "Disjoint paths in expander graphs via random walks: A short survey," in *Proc. Int. Workshop Randomization Approximation Techn. Comput. Sci.*, 1998, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=646975.711409

[24] J. Kleinberg and R. Rubinfeld, "Short paths in expander graphs," in *Proc. IEEE Symp. Found. Comput. Sci.*, Oct. 1996, pp. 86–95.

[25] H. Okamura, "Multicommodity flows in planar graphs," *J. Combinat. Theory, Ser. B*, vol. 31, no. 1, pp. 75–81, Aug. 1981.

[26] C. Barnhart, C. A. Hane, and P. H. Vance, "Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems," *Oper. Res.*, vol. 48, no. 2, pp. 318–326, 2000.

[27] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[28] N. Foster, "The frenetic network controller," in *Proc. OCaml Users Developers Workshop*, 2013, pp. 1–2.

[29] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comp. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[30] *Puppet*. Accessed: Dec. 6, 2017. [Online]. Available: http://puppetlabs.com

[31] 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. Symp. Netw. Syst. Desing Implement.*, 2014, pp. 533–546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616497

[32] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA, USA: Addison-Wesley, 1979.

[33] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Symp. Oper. Syst. Desing Implement.*, 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/full_papers/dean/dean.pdf

[34] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi, "Detailed analysis of skype traffic," *IEEE Trans. Multimedia*, vol. 11, no. 1, pp. 117–127, Jan. 2009.

[35] V. Jeyakumar *et al.*, "EyeQ: Practical network performance isolation at the edge," in *Proc. Symp. Netw. Syst. Desing Implement.*, 2013, pp. 297–312. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482656

[36] Y. Vigfusson *et al.*, "Dr. Multicast: Rx for data center communication scalability," in *Proc. EuroSys*, 2010, pp. 349–362, doi: 10.1145/1755913.1755949.

[37] Gurobi Optimization Inc., *The Gurobi Optimizer*. Accessed: Dec. 6, 2017. [Online]. Available: http://www.gurobi.com

[38] L. D. Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[39] P. Hooimeijer. *Dprle Decision Procedure Library*. Accessed: Dec. 6, 2017. [Online]. Available: http://www.cs.virginia.edu/~ph4u/dprle/

[40] *Automatic Test Packet Generation*. Accessed: Dec. 6, 2017. [Online]. Available: https://github.com/eastzone/atpg

[41] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 164–206, 2003.

[42] R. Subramaniyan, P. Raman, A. D. George, M. A. Radlinski, and M. A. Radlinski, "GEMS: Gossip-enabled monitoring service for scalable heterogeneous distributed systems," *Cluster Comput.*, vol. 9, no. 1, pp. 101–120, 2006.

[43] M. Jelasity, A. Montresor, and O. Babaoglu, "T-Man: Gossip-based fast overlay topology construction," *Comput. Netw.*, vol. 53, no. 13, pp. 2321–2339, 2009.

[44] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. Symp. Oper. Syst. Princ.*, 2007, pp. 205–220.

[45] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[46] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[47] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. Symp. Oper. Syst. Desing Implement.*, 2006, pp. 335–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487

[48] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proc. Symp. Oper. Syst. Princ.*, 2011, pp. 15–28, doi: 10.1145/2043556.2043559.

[49] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. Symp. Oper. Syst. Desing Implement.*, 2012, pp. 251–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905

[50] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun./Jul. 2010, pp. 527–536.

[51] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998, doi: 10.1145/279227.279229.

[52] C. Dixon *et al.*, "ETTM: A scalable fault tolerant network manager," in *Proc. Symp. Netw. Syst. Desing Implement.*, 2011, pp. 85–98. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972467

[53] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2009, pp. 51–62, doi: 10.1145/1592568.1592576.

[54] *The Internet Topology Zoo*. Accessed: Dec. 6, 2017. [Online]. Available: http://www.topology-zoo.org

[55] NetworkX. *NetworkX developers*. Accessed: Dec. 6, 2017. [Online]. Available: https://networkx.github.io

[56] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2008, pp. 63–74.

[57] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2011, pp. 242–253, doi: 10.1145/2018436.2018465.

[58] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks," in *Proc. Workshop Hot Topics Cloud Comput.*, 2010, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863104

[59] L. Popa *et al.*, "FairCloud: Sharing the network in cloud computing," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 187–198, doi: 10.1145/2342356.2342396.

[60] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proc. Conf. Comput. Commun. Secur.*, 2000, pp. 190–199, doi: 10.1145/352600.353052.

[61] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proc. Conf. Syst. Admin.*, 1999, pp. 229–238.

[62] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica, "Pathlet routing," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 111–122, 2009, doi: 10.1145/1594977.1592583.

[63] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. Workshop Res. Enterprise Netw.*, 2009, pp. 1–10.

[64] J. Sherry *et al.*, "Making middleboxes someone else's problem: network processing as a cloud service," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 13–24, doi: 10.1145/2377677.2377680.

[65] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. Symp. Oper. Syst. Princ.*, 2015, pp. 121–136, doi: 10.1145/2815400.2815423.

[66] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2013, pp. 327–338.

[67] S. Chong *et al.*, "Secure Web Applications via Automatic Partitioning," in *Proc. Symp. Oper. Syst. Princ.*, 2007, pp. 31–44, doi: 10.1145/1323293.1294265.

[68] J. Liu *et al.*, "Fabric: A platform for secure distributed computation and storage," in *Proc. ACM SIGOPS Eur. Workshop*, 2009, pp. 321–334, doi: 10.1145/1629575.1629606.

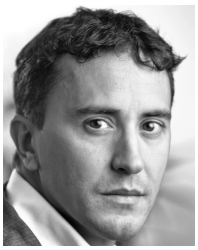
Robert Soulé received the B.A. degree from Brown University, Providence, RI, USA, in 1999, and the Ph.D. degree from New York University, New York, NY, USA, in 2012. He is currently an Assistant Professor at the Università della Svizzera italiana and a Research Scientist at Barefoot Networks. Prior to joining USI, he was a Post-Doctoral Associate at Cornell University, Ithaca, NY, USA. For two years, he was a research co-op with the Data Intensive Systems and Analytics Group, IBM T. J. Watson Research Center. His research interests are in distributed systems, networking, and applied programming languages.

Shrutarshi Basu is currently a Post-Doctoral Fellow in computer science at Harvard University, Cambridge, MA, USA. He is involving in the legal applications and implications of computational technologies, particularly programming languages and associated tools and techniques. Previously, he applied techniques and tools from programming languages to software-defined networks and optical networks.

Parisa Jalili Marandi received the Ph.D. degree in computer science from Università della Svizzera italiana, Switzerland, in 2014. She is currently an Applied Researcher at Microsoft Research Cambridge, Cambridge, U.K. She is interested in the theory and practice of large scale distributed systems, cloud computing, and resource optimization.

Fernando Pedone received the Ph.D. degree from EPFL in 1999. He is currently a Full Professor with the Faculty of Informatics, Università della Svizzera italiana, Switzerland. He has been also with Cornell University, Ithaca, NY, USA, as a Visiting Professor, EPFL, and Hewlett-Packard Laboratories. He has authored more than 100 scientific papers and six patents. He is a co-editor of the book *Replication: theory and practice*. His research interests include the theory and practice of distributed systems and distributed data management systems.

Robert Kleinberg is currently an Associate Professor of computer science at Cornell University, Ithaca, NY, USA. His research pertains to the design and analysis of algorithms, and their applications to machine learning, economics, networking, and other areas. Prior to receiving his doctorate from MIT in 2005, he spent three years at Akamai Technologies, where he assisted in designing the world's largest Internet CDN. He was a recipient of the Microsoft Research New Faculty Fellowship, the Alfred P. Sloan Foundation Fellowship, and the NSF CAREER Award.



Emin Gün Sirer is the Co-Director of the Initiative for Cryptocurrencies and Smart Contracts and an Associate Professor of computer science at Cornell University, Ithaca, NY, USA. His research interests span distributed systems, operating systems, and software infrastructure for large scale services. He is known for his fundamental contributions to fintech and the science behind blockchain technologies.

Nate Foster received the B.A. degree in computer science from the Williams College, the M.Phil. degree in history and philosophy of science from Cambridge University, and the Ph.D. degree in computer science from the University of Pennsylvania, Philadelphia, PA, USA. He is currently an Associate Professor of computer science at Cornell University, Ithaca, NY, USA, and a Principal Research Engineer at Barefoot Networks. The goal of his research is to develop languages and tools that make it easy for programmers to build secure and reliable systems. His current work focuses on the design and implementation of languages for programming software-defined networks. He received several awards, including the Sloan Research Fellowship and the NSF CAREER Award.