# Merlin: A Language for Provisioning Network Resources

Robert Soulé* Shrutarshi Basu† Parisa Jalili Marandi* Fernando Pedone*
Robert Kleinberg† Emin Gün Sirer† Nate Foster†
*University of Lugano †Cornell University



## ABSTRACT

This paper presents Merlin, a new 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 Merlin compiler uses a combination of advanced techniques to translate these policies into code that can be executed on network elements including a constraint solver that allocates bandwidth using parameterizable heuristics. To facilitate dynamic adaptation, 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 scalability of Merlin on real-world topologies and applications. Overall, Merlin simplifies network administration by providing high-level abstractions for specifying network policies and scalable infrastructure for enforcing them.

## 1. INTRODUCTION

Network operators today must deal with a wide range of management challenges from increasingly complex policies to a proliferation of heterogeneous devices to ever-growing traffic demands. Software-defined networking (SDN) provides tools that could be used to address these challenges, but existing SDN APIs are either too low-level or too limited in functionality to enable effective implementation of rich network-wide policies. As a result, there is widespread interest in academia and industry in higher-level programming languages and "northbound APIs" that provide convenient control over the complete set of resources in a network.

Unfortunately, despite several notable advances, there is still a large gap between the capabilities of existing SDN APIs and the realities of network management. Current programming languages focus mostly on packet forwarding and largely ignore functionality such as bandwidth and richer packet-processing functions that can only be implemented on middleboxes, end hosts, or with custom hardware [19, 47, 64, 2, 50]. Network orchestration frameworks provide powerful mechanisms that handle a larger set of concerns including middlebox placement and bandwidth [22, 34, 54, 57], but they expose APIs that are either extremely simple (e.g., sequences of middleboxes) or not specified in detail. Overall, the challenges of managing real-world networks using existing SDN APIs remain unmet.

This paper presents a new SDN programming language that is designed to fill this gap. This language, called Merlin, provides a collection of high-level programming constructs for (i) classifying packets; (ii) controlling forwarding paths; (iii) specifying rich packet transformations; and (iv) provisioning bandwidth in terms of maximum limits and minimum guarantees. These features go far beyond what can be realized just using SDN switches or with existing languages like Frenetic [19], Pyretic [46], and Maple [64]. As a result, implementing Merlin is non-trivial because it involves determining allocations of limited network-wide resources such as bandwidth—the simple compositional translations used in existing SDN compilers cannot be readily extended to handle the new features provided in Merlin.

Merlin uses a variety of compilation techniques to determine forwarding paths, map packet transformations to network elements, and allocate bandwidth. These techniques are based on a unified representation of the physical network topology and the constraints expressed by the policy—a logical topology. For traffic with bandwidth guarantees, the compiler uses a mixed integer program formulation to solve a variant of the multi-commodity flow constraint problem. For traffic without bandwidth guarantees, 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 kinds of traffic separately allows Merlin to provide a uniform interface to programmers while reducing the size of the more expensive constraint problems that must be solved. The compiler also handles generation of low-level instructions for a variety of 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

$$
\begin{array}{rcll}
loc & \in & \textit{Locations} & \\
t & \in & \textit{Packet transformations} & \\
pol & ::= & [s_1; \ldots ; s_n], \phi & \text{Policies} \\
s & ::= & id : p \rightarrow r & \text{Statements} \\
\phi & ::= & \mathsf{max}(e, n) \mid \mathsf{min}(e, n) & \text{Presburger Formulas} \\
 & \mid & \phi_1 \;\mathsf{and}\; \phi_2 \mid \phi_1 \;\mathsf{or}\; v_2 \mid !\, \phi_1 & \\
e & ::= & n \mid id \mid e + e & \text{Bandwidth Terms} \\
a & ::= & . \mid c \mid a\, a \mid a \,|\, a \mid a^* \mid !\, a & \text{Path Expression} \\
p & ::= & m \mid p_1 \;\mathsf{and}\; p_2 \mid p_1 \;\mathsf{or}\; p_2 \mid !\, p_1 & \text{Predicates} \\
 & \mid & h.f = n \mid \mathsf{true} \mid \mathsf{false} & \\
c & ::= & loc \mid t & \text{Path Element}
\end{array}
$$

**Figure 1: Merlin abstract syntax.**

modifications made by other negotiators do not lead to policy violations. Again, the use of a high-level programming language is essential, as it provides a concrete basis for analyzing, transforming, and verifying policies.

We have built a working prototype of Merlin, and used it to implement a variety of practical policies that demonstrate the expressiveness of the language. These examples illustrate how Merlin supports a wide range of network functionality including simple forwarding policies, policies that require rich transformations usually implemented on middleboxes such as load balancing and deep-packet inspection, and policies that provide bandwidth guarantees. We have also implemented negotiators that realize max-min fair sharing and additive-increase multiplicative-decrease dynamic allocation schemes. Our experimental evaluation shows that the Merlin compiler can quickly provision and configure 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:

- The design of high-level network management abstractions realized in an expressive policy language that models packet classification, forwarding, transformation, and bandwidth.
- Compilation algorithms based on a translation from policies to constraint problems that can be solved using mixed integer programming.
- An approach for dynamically adapting policies using negotiators and accompanying verification techniques, made possible by the language design.

The rest of this paper describes the design of the Merlin language (§2), compiler (§3), and runtime transformations (§4). It then describes the implementation (§5) and presents the results from our performance evaluation (§6).

# 2. LANGUAGE DESIGN

The Merlin policy language is designed to give programmers a rich collection of constructs that allow them to specify the intended behavior of the network at a high level of abstraction. As an example, suppose that we want to place a bandwidth cap on FTP control and data transfer traffic, while providing a bandwidth guarantee to HTTP traffic. The program below realizes this specification using a sequence of Merlin policy statements, followed by a logical formula. Each statement contains a variable that tracks the amount of bandwidth used by packets processed with that statement, a predicate on packet headers that identifies a set of packets, and a regular expression that describes a set of forwarding paths through the network:

```
[ x : (eth.src = 00:00:00:00:00:01 and
       eth.dst = 00:00:00:00:00:02 and
       tcp.dst = 20) -> .* dpi .*
  y : (eth.src = 00:00:00:00:00:01 and
       eth.dst = 00:00:00:00:00:02 and
       tcp.dst = 21) -> .*
  z : (eth.src = 00:00:00:00:00:01 and
       eth.dst = 00:00:00:00:00:02 and
       tcp.dst = 80) -> .* dpi *. nat .* ],
  max(x + y,50MB/s) and min(z,100MB/s)
```

The statement on the first line asserts that FTP traffic from the host with MAC address `00:00:00:00:00:01` to the host with MAC address `00:00:00:00:00:02` 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 and HTTP traffic between the same hosts respectively. Note that the statement for FTP control traffic does not include any constraints on its forwarding path, while the HTTP statement includes both a `dpi` and a `nat` constraint. The formula on the last line declares a bandwidth cap (`max`) on the FTP traffic, and a bandwidth guarantee (`min`) for the HTTP traffic. The rest of this section describes the constructs used in this policy in detail.

## 2.1 Syntax and semantics

The syntax of the Merlin policy language is defined by the grammar in Figure 1. A policy is a set of *statements*, each of which specifies the handling of a subset of traffic, together with a *logical formula* that expresses a global bandwidth constraint. For simplicity, we require that the statements have disjoint predicates and together match all packets. In our implementation, these requirements are enforced by a simple pre-processor. Each policy statement comprises several components: an *identifier*, a *logical predicate*, and a *regular expression*. The identifier provides a way to identify the set of packets matching the predicate, while the regular expression specifies the forwarding paths and packet transformations that should be applied to matching packets. Together, these abstractions facilitate thinking of the network as a "big switch" [35], while enabling programmers to retain precise control over forwarding paths and bandwidth usage.

*Logical predicates.* Merlin supports a rich predicate language for classifying packets. Atomic predicates of the form $h.f = n$ denote the set of packets whose header field $h.f$ is equal to $n$. For instance, in the example policy above, statement `z` contains the predicate that matches packets with eth source address 00:00:00:00:00:01, destination address 00:00:00:00:00:02, and tcp port 80. Merlin provides atomic predicates for a number of standard protocols including Ethernet, IP, TCP, and UDP, and a special predicate for matching packet payloads. Predicates can also be combined using conjunction (and), disjunction (or), and negation (!).

*Regular expressions.* Merlin allows programmers to specify the set of allowed forwarding paths through the network using regular expressions—a natural and well-studied mathematical formalism for describing paths through a graph (such as a finite state automaton or a network topology). However, rather than matching strings of characters, as with ordinary regular expressions, Merlin regular expressions match sequences of network locations (including names of transformations, as described below). The compiler is free to select any matching path for forwarding traffic as long as the other constraints expressed by the policy are satisfied. We assume that the set of network locations is finite. As with POSIX regular expressions, the dot symbol (.) matches a single element of the set of all locations.

*Packet transformations.* Merlin regular expressions may also contain names of packet-processing functions that may transform the headers and contents of packets. Such functions can be used to implement a variety of useful operations including deep packet inspection, network address translation, wide-area optimizers, caches, proxies, traffic shapers, and others. The compiler determines the location where each function is enforced, using a mapping from function names to possible locations supplied as a parameter. The only requirements on these functions are that they must take a single packet as input and generate zero or more packets as output, and they must only access local state. In particular, the restriction to local state allows the compiler to freely place functions without having to worry about maintaining global state. Merlin's notion of packet processing functions that can be easily moved within the network is similar in spirit to network function virtualization [9].

*Bandwidth constraints.* Merlin policies use formulas in Presburger arithmetic (i.e, first-order formulas with addition but without multiplication, to ensure decidability) to specify constraints that either limit (max) or guarantee (min) bandwidth. 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, and policies that lack a maximum rate may send traffic at rates up to line speed. Bandwidth limits and guarantees differ from packet processing functions in one important aspect: they represent an explicit allocation of global network resources. Hence, additional care is needed when compiling them.

*Syntactic sugar.* In addition to the core constructs shown in Figure 1, Merlin also supports several forms of syntactic sugar that can simplify the expression of complex policies. Notably, Merlin provides set literals and functions for operating on and iterating over sets. For example, the following policy,

```
srcs := {00:00:00:00:00:01}
dsts := {00:00:00:00:00:02}
foreach (s,d) in cross(srcs,dsts):
  tcp.dst = 80 ->
  ( .* nat .* dpi .*) at max(100MB/s)
```

is equivalent to statement z from the 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 src s, destination d, and term tcp.dst = 80.

*Summary.* Overall, Merlin's policy language enables direct expression of high-level network policies. Crucial to its design, however, is that policies can be distilled into their component constructs, which can then be distributed across many devices in the network to collectively enforce the global policy. The subsequent sections present these distribution and enforcement mechanisms in detail.

# 3. COMPILER

The Merlin compiler performs three essential tasks: (i) it translates global policies into locally-enforceable policies; (ii) it determines the paths used to carry traffic across the network, places packet transformations on middleboxes and end hosts, and allocates bandwidth to individual flows; and (iii) it generates low-level instructions for network devices and end hosts. To do this, the compiler takes as inputs the Merlin policy, a representation of the physical topology, and a mapping from transformations to possible placements, and builds a logical topology that incorporates the structure of the physical topology as well as the constraints encoded in the policy. It then analyzes the logical topology to determine allocations of resources and emits low-level configurations for switches, middleboxes, and end hosts.

## 3.1 Localization

Merlin's Presburger arithmetic formulas are an expressive way to declare bandwidth constraints, but actually implementing them leads to several challenges: aggregate guarantees can be enforced using shared quality-of-service queues on switches, but aggregate bandwidth 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 the 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

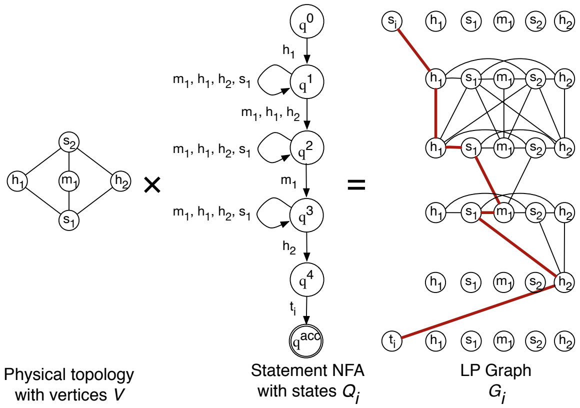



**Figure 2: Logical topology for the example policy. The thick, red path illustrates a solution.**

permissible. For example, the `max` term in the running example would be localized to `max(x, 25MB/s) and max(y, 25MB/s)`. Rewriting policies in this way involves an inherent tradeoff: localized enforcement increases scalability, but risks underutilizing resources if the static allocations do not reflect actual usage. In Section 4, we describe how Merlin navigates this tradeoff via a run-time mechanism, called *negotiators*, that can dynamically adjust allocations.

## 3.2 Provisioning for Guaranteed Rates

The most challenging aspect of the Merlin compilation process is provisioning bandwidth for traffic with guarantees. To do this, the compiler encodes the input policy and the network topology into a constraint problem whose solution, if it exists, can be used to determine the configuration of each network device.

*Logical topology.* Recall that a statement in the Merlin language contains a regular expression, which constrains the set of forwarding paths and packet processing functions that may be used to satisfy the statement. To facilitate the search for routing paths that satisfy these constraints, the compiler represents them internally as a directed graph $\mathcal{G}$ whose paths correspond to physical network paths that respect the path constraints of a single statement. The overall graph $\mathcal{G}$ for the policy is a union of disjoint components $\mathcal{G}_i$, one for each statement $i$.

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 map $a_i$ into a regular expression $\bar{a}_i$ over the set of locations using a simple substition: for every occurrence of a packet transformation, 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 network address translation, then the regular expression `.* nat .*` would be transformed to `.* (h1|h2|m1) .*`. The next step is to transform the regular expression $\bar{a}_i$ into a nondeterministic finite automaton (NFA), denoted $\mathcal{M}_i$, that accepts the set of strings in the regular language given by $\bar{a}_i$.

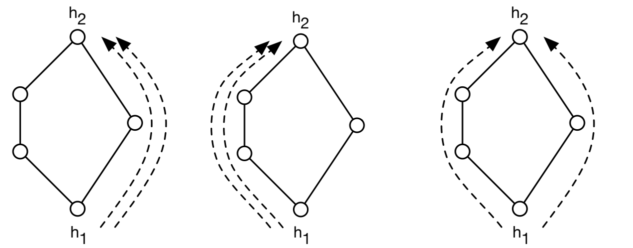
Letting $L$ denote the set of locations in the physical network and $\mathcal{Q}_i$ denote the state set of $\mathcal{M}_i$, the vertex set of $\mathcal{G}_i$ is the Cartesian product $L \times \mathcal{Q}_i$ together with two special vertices $\{s_i, t_i\}$ that serve as a universal source and sink 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 $i$:

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 $q_1, \ldots, q_k$.*

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 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. $\square$

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, one sees that all edges between these two rows lead into node $(m_1,q^3)$. This, in turn, means that 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* [1], with two additional types of constraints: (i) *integrality constraints* demand that only one path may be selected for each statement,




(a) Shortest-Path (b) Min-Max Ratio (c) Min-Max Reserved

**Figure 3: Path selection heuristics.**

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 into multi-commodity flow problems renders them NP-complete in the worst case, but a number of approaches have been developed over the years for surmounting this problem, ranging from approximation algorithms [8, 11, 15, 37, 39], to specialized algorithms for topologies such as expanders [6, 21, 36] and planar graphs [51], to the use of mixed-integer programming [5]. Our current implementation adopts the latter technique.

Our mixed-integer program (MIP) has one $\{0, 1\}$-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

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

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 $\mathcal{G}_i$.

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

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

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

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

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 route assignment will not exceed the capacity of any link, by asserting that the fraction of capacity reserved on any link cannot 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 assigned 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 as longer paths tend to experience increased 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 the network 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 disrupted by a single link failure.

The differences between these heuristics are illustrated in Figure 3 which depicts a simple network with hosts `h1` and `h2` connected by a pair of disjoint paths. The left path comprises three edges of capacity 400MB/s. The right path comprises two edges of capacity 100MB/s. The figure shows the paths selected for two statements each requesting a bandwidth guarantee of 50MB/s. Depending on the heuristic, 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).

## 3.3 Provisioning for Best-Effort Rates

For traffic requiring 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. A sink-tree for a particular network node forwards traffic from elsewhere on the network to that node. Merlin does this by computing the cross product of the regular expression NFA and the network topology representation, as just described, and then performing a breath-first search over the resulting graph. To further improve scalability, the compiler uses a small optimization: it works on a topology that only includes 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.

## 3.4 Code Generation

Next, the Merlin compiler generates code to enforce the policy using the devices available in the network. The actual code is determined both by the requested functionality and the type of target device. For basic forwarding and bandwidth guarantees, Merlin generates instructions for OpenFlow switches and controllers [45] that install forwarding rules and configure port queues. For packet transformations such as deep packet inspection, network address translation, and intrusion detection, Merlin generates scripts to install and configure middleboxes, such as Click [38]. Traffic filtering and rate limiting are implemented by generating calls to the standard Linux utilities `iptables`. Of course, other approaches are possible. For example, Merlin could implement packet transformations by using Puppet [53] to provision virtual machines that implement those transformations.

Because Merlin controls forwarding paths but also supports middleboxes that may modify headers (such as NAT boxes), the compiler needs to use a forwarding mechanism that is robust to changes in packet headers. Our current implementation uses VLAN tags to encode paths to destination switches, one tag per sink tree. All packets destined for that tree's sink are tagged with a tag 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 a unique host identifier (e.g., the MAC address) is used to forward traffic to the appropriate host. This approach is similar to the technique used in FlowTags [17].

Merlin can provide greater flexibility and expressiveness by directly generating packet-processing code, which can be executed by an interpreter running on end hosts or on middleboxes. We have also built a prototype that runs as a Linux kernel module and uses the `netfilter` callback functions to access packets on the network stack. The interpreter accepts and enforces programs that can filter or rate limit traffic using a richer set of predicates than those offered by `iptables`. It is designed to have minimal dependencies on operating system services in order to make it portable across different systems. The current implementation requires only about a dozen system calls to be exported from the operating system to the interpreter. In on-going work, we are exploring additional functionality, with the goal or providing a general runtime as the target for the Merlin complier. However, using end hosts assumes a trusted deployment in which all host machines are under administrative control. An interesting, but orthogonal, problem is to deploy Merlin in an untrusted environment. Several techniques have been proposed to verify that an untrusted machine is running certain software. Notable examples include proof carrying code [49], and TPM-based attestations [16, 60].

Merlin can be instantiated with a variety of backends to capitalize on the capabilities of the available devices in the network. Although the expressiveness of policies is bounded by the capabilities of the devices, Merlin provides a unified interface for programming them.

# 4. DYNAMIC ADAPTATION

The Merlin compiler described in the preceding section translates policies into static configurations. Of course, these static configurations may under-utilize resources, depending on how traffic demands evolve over time. Moreover, in a shared environment, network tenants may wish to customize global policies to suit their own needs—e.g., to add additional security constraints.

To allow for the dynamic modification of policies, Merlin uses small run-time components called *negotiators*. Negotiators are policy transformers and verifiers—they allow policies to be delegated to tenants for modification and they provide a mechanism for verifying that modifications made by tenants do not lead to violations of the original global policy. Negotiators depend critically on Merlin's language-based approach. The same abstractions that allow policies to be mapped to constraint problems (i.e., predicates, regular expressions, and explicit bandwidth reservations), 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.

## 4.1 Transformations

With negotiators, tenants can transform global network policies by *refining* the delegated policies to suit their own demands. Tenants may modify policies in three ways: (i) policies may be refined with respect to packet classification; (ii) forwarding paths may be further constrained; and (iii) bandwidth allocations may be revised.

*Refining policies.* 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 ones that match HTTP traffic and all other traffic:

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

The partitioning must be total—all packets identified by the original policy must be identified by the set of new policies.

*Constraining paths.* Merlin programmers declare path constraints using regular expressions that match sequences of network locations or packet transformations. 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 modified to say that the traffic must additionally pass through a DPI function:

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

*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. The requirement for a valid transformation is that the sum of the new allocations must 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 100MB/s:

```
[x : (ip.src = 192.168.1.1 and
      ip.dst = 192.168.1.2) -> .*],
max(x, 100MB/s)
```

This policy could be modified as follows:

```
[x : (ip.src = 192.168.1.1 and
      ip.dst = 192.168.1.2 and
      tcp.dst = 80) -> .* log .*],
[y : (ip.src = 192.168.1.1 and
      ip.dst = 192.168.1.2 and
      tcp.dst = 22) -> .* ],
[z : (ip.src = 192.168.1.1 and
      ip.dst = 192.168.1.2 and
      !(tcpDst=22|tcpDst=80)) -> .* dpi .*],
max(x, 50MB/s)
and max(y, 25MB/s)
and max(z, 25MB/s)
```

It gives 50MB/s to HTTP traffic with the constrain that it passes through a `log` that monitors all web requests, 25MB/s to SSH traffic, and 25MB/s to the remaining traffic, which must flow through a `dpi` box.

## 4.2 Verification

Allowing tenants to make arbitrary modifications to policies would not be safe. 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 simply has to check that for every statement in the original policy, the set of paths allowed for matching packets in the refined policy is included in the set of paths in the original, and the bandwidth constraints in the refined policy imply the bandwidth constraints in the original. These conditions can be decided using a simple algorithm that performs a pairwise comparison of all statements in the original and modified policies, (i) checking for language inclusion [28] between the regular expressions in statements with overlapping predicates, and (ii) checking that the sum of the bandwidth constraints in all overlapping predicates implies the original constraint.

## 4.3 Adaptation

Bandwidth re-allocation does not require recompilation of the global policy, and can thus happen quite rapidly. As a proof-of-concept, we implemented negotiators that can provide both min-max fair sharing and additive-increase, multiplicative decrease allocation schemes. These negotiators allow traffic policies to change with dynamic workloads, while still obeying the overall static global policies. Changes in path constraints require global recompilation and updating forwarding rules on the switches, so they incur a greater overhead. However, we believe these changes are likely to occur less frequently than changes to bandwidth allocations.

# 5. IMPLEMENTATION

We have implemented a full working prototype of the Merlin system in OCaml and C. Our implementation uses the Gurobi Optimizer [25] to solve constraints, the Frenetic controller [20] to install forwarding rules on OpenFlow switches, the Click router [38] 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. It would be easy to instantiate our design with other systems, and our implementation provides a clean interface for incorporating additional backends.

Our implementation of Merlin negotiator and verification mechanisms leverages standard algorithms for transforming and analyzing predicates and regular expressions. To delegate a policy, Merlin simply intersects the predicates and regular expressions in each statement the original policy to

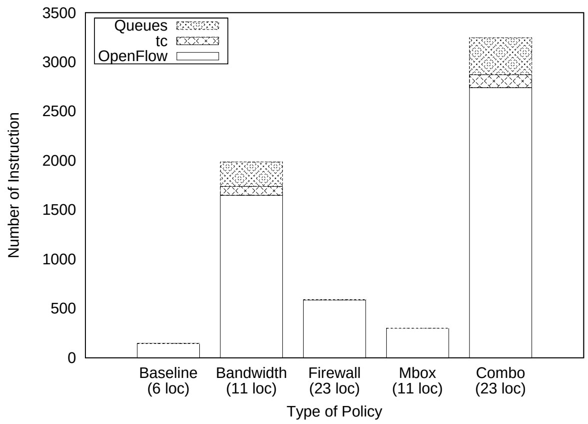



**Figure 4: Merlin expressiveness, measured using policies for the Stanford campus network topology.**

project out the policy for the sub-network. To verify implications between policies, Merlin uses the Z3 SMT solver [48] to check predicate disjointness, and the Dprle library [27] to check inclusions between regular expressions.

# 6. 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 (§6.2) 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 experimental evaluation shows that Merlin can quickly provision and configure real-world datacenter and enterprise networks, that Merlin can be used to obtain better performance for big-data processing applications and replication systems, and that Merlin allows administrators to succinctly express network policies.

## 6.1 Expressiveness.

To explore the expressiveness of the Merlin policy languages, we built several network policies for the 16-node Stanford core campus network topology [3]. We created 24 subnets and 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 real-world network.

The Merlin policies we implemented are as follows:

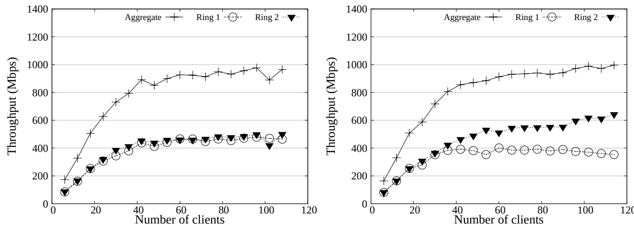
1. *All-pairs connectivity*. This policy implements basic forwarding between all pairs of hosts and provides a baseline measurement of the number of low-level instructions that would be needed in almost any non-trivial application. The Merlin policy is only 6 lines long and compiles to 145 OpenFlow rules.
2. *Bandwidth caps and guarantee*. 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 would 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 configurations. 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 connected to the network ingress switches. The baseline policy is altered to forward all packets matching a particular pattern (e.g., `tcp.dst = 80`) through the middlebox. This policy requires 23 lines of Merlin code, but generates over 500 OpenFlow instructions.
4. *Monitoring middlebox*. 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 sets must be passed 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 number.
5. *Combination*. This policy augments the basic connectivity with a filter for web traffic, a bandwidth guarantee for certain traffic classes and an inspection policy for a certain class of hosts. This policy requires 23 lines of Merlin code, but generates over 3000 low-level instructions.

The results of this experiment are depicted in Figure 4. Overall, it shows that 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.

## 6.2 Application Performance

Our second experiment measured Merlin's ability to improve end-to-end performance for real-world applications.

*Hadoop*. Hadoop is a popular open-source MapReduce [13] implementation, and is widely-used for data analytics. A



(a) (b)

**Figure 5: Ring-Paxos (a) without and (b) with Merlin.**

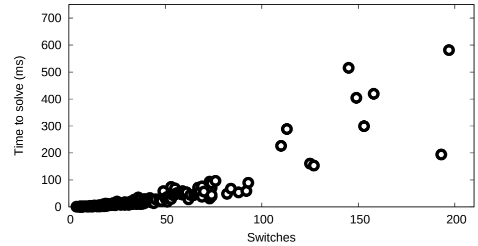



**Figure 6: Compilation times for Internet Topology Zoo.**

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 use UDP-based gossip protocols to update state, such as system monitoring tools [63, 62], network overlay management [32], and even distributed storage systems [63, 14]. A sensible network policy would be to provide guaranteed bandwidth to Hadoop jobs so that they finish expediently, and give the UDP traffic only best-effort guarantees.

With Merlin, we implemented this policy using just three policy statements. To show the impact of the policy, we ran a Hadoop job that sorts 10GB of data, and measured the time to complete it on a cluster of four servers, under three different configurations:

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

The measurements demonstrate the expected results. 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 the Merlin policy providing 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 [41, 56] at the core of many current systems (e.g., Googles Chubby [7], Scatter [23], Spanner [12]). 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 the replicas: (i) every nonfaulty replica must receive all commands in the same order; and (ii) the execution of commands must be deterministic.

Because ordering commands in a distributed setting is a non-negligible operation, 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, the service state can be partitioned and each partition replicated individually (e.g., by separating data from meta-data), but the partitions will compete for shared resources (e.g., common nodes and network links).

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 [44], a highly efficient implementation of the Paxos protocol [42]. 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 5 (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 5 (b), we provide a bandwidth guarantee for Service 2. Note that this guarantee does not come at the expense of utilization. If Service 2 stops sending traffic, Service 1 is free to use the available bandwidth.

*Summary.* Overall, these experiments show that Merlin policies 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.

## 6.3 Compilation and Verification

The scalability of the Merlin compiler and verification framework depend on both the size of the network topology and the number of traffic classes. Our third experiment evaluates the scalability of Merlin under a variety of scenarios.

*Compiler.* We first measured the compilation time needed by Merlin to provide pair-wise connectivity between all hosts in a topology. This task, which could be computed offline, has been used to evaluate other systems, including VMware's NSX, which reports approximately 30 minutes to achieve 100% connectivity from a cold boot [40]. We used the Internet Topology Zoo [29] dataset, which contains 262 topologies that represent a large diversity of network structures.

| Traffic Classes | Hosts | Switches | LP construction (ms) | LP solution (ms) | Rateless 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 |

**Figure 7: Number of traffic classes, topology sizes, and solution times for fat tree topologies with 5% of the traffic classes with guaranteed bandwidth.**

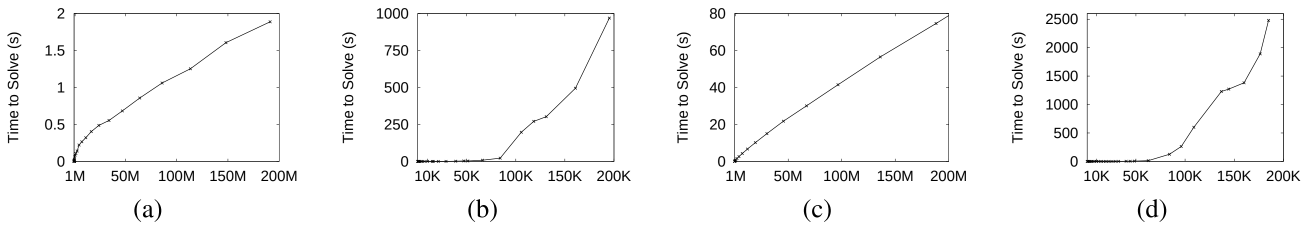



**Figure 8: Compilation times for an increasing number of traffic classes for (a) all pairs connectivity on a balanced tree, (b) 5% of the traffic with guaranteed priority on a balanced tree, (c) all pairs connectivity on a fat tree, (d) 5% of the traffic with guaranteed priority on a fat tree.**

The topologies have average size of 40 switches, with a standard deviation of 30 switches. The Merlin compiler takes less than 50ms to provide connectivity for the majority of topologies, and less than 600ms to provide connectivity for all but one of the topologies. Figure 6 shows the results for 261 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.

To explore the scalability with bandwidth guarantees, we measured the compilation time on two types of tree topologies, balanced and fat trees, for an increasing number of traffic classes. Each traffic class represents a unidirectional stream going from one host at the edge of the network to another. Thus, the total number of classes correspond to the number of point-to-point traffic flows. Table 7 shows a sample of topology sizes and solution times for various traffic classes for fat tree topologies. For both types of topologies, we took two sets of measurements: the time to provide pair-wise connectivity with no guarantees, and the time to provide connectivity when 5% of the traffic classes receive guarantees. Figure 8 shows the results. As expected, providing bandwidth guarantees adds overhead to the compilation time. For the worst case scenario that we measured, on a network with $400,000$ total traffic classes, with $20,000$ of those classes receiving bandwidth guarantees, Merlin took around 41 minutes to find a solution. To put that number in perspective, B4 [31] only distinguishes 13 traffic classes. Merlin finds solutions for 100 traffic classes with guarantees in a network with 125 switches in less than 5 seconds.

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 9, 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

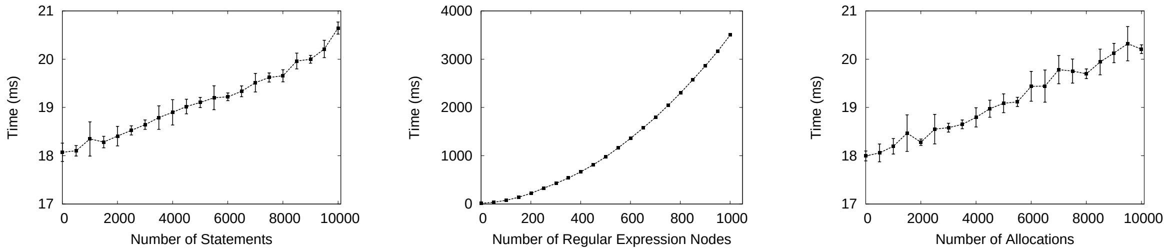
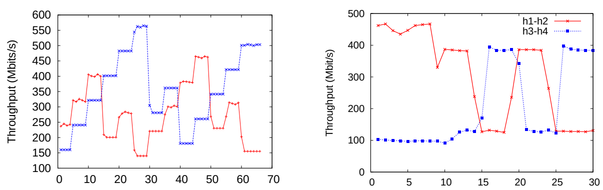



**Figure 9: Time taken to verify a delegated policy for an increasing number of delegated predicates, increasingly complex regular expressions, and an increasing number of bandwidth allocations.**




(a) (b)

**Figure 10: (a) AIMD and (b) MMFS dynamic adaptation.**

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). With AIMD, tenants adjust resource demands by incrementally trying to increasing their allocation. With MMFS, tenants declare resource requirements ahead of time. The negotiator attempts to satisfy demands starting with the smallest. Remaining bandwidth is distributed among all tenants. Figure 10 (a) shows the bandwidth usage over time for two hosts using the AIMD strategy. Figure 10 (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 graphs were generated using our hardware testbed. Overall, the negotiators allow the network to quickly adapt to changing resource demands, while respecting the global constraints imposed by the policy.

## 7. RELATED WORK

In prior work [61], we presented a preliminary design for Merlin including sketching the encoding of path selection as a constraint problem, and presenting ideas for language-based delegation and verification. This paper expands our eariler work with a complete description of Merlin's design and implementation and an experimental evaluation.

A number of systems in recent years have investigated mechanisms for providing bandwidth caps and guarantees [4, 59, 52, 33], implementing traffic filters [30, 55], or specifying forwarding policies at different points in the network [19, 24, 46, 26]. Merlin builds on these approaches by providing a unified interface and central point of control for switches, middleboxes, and end hosts.

SIMPLE [54] 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. APLOMB [58] is a system that allows middleboxes to be obtained from a cloud service. Merlin is designed to provide a programming abstraction for the whole network, of which a middlebox is just one kind of network element. It would be interesting to extend Merlin with a back-end that supports cloud-hosted middleboxes as in APLOMB.

Many different programming languages have been proposed in recent years including Frenetic [19], Pyretic [47], and Maple [64]. 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 [18] system, which allows end hosts to make explicit requests for network resources like bandwidth. Unlike Merlin, PANE does not provide mechanisms for partitioning functionality across a variety of devices and delegation is supported at the level of individual network flows, rather than entire policies.

The Merlin compiler implements a form of program partitioning. This idea has been previously used in other domains including secure web applications [10], and distributed computing and storage [43].

## 8. CONCLUSION

The success of programmable network platforms has demonstrated the benefits of high-level languages for managing networks. Merlin complements these approaches by further raising the level of abstraction. Merlin allows administrators to specify the functionality of an entire network, leaving the low-level configuration of individual components to the compiler. At the same time, Merlin provides tenants with the freedom to tailor policies to their particular needs, while assuring administrators that the global constraints are correctly enforced. Overall, this approach significantly simplifies network administration, and lays a solid foundation for a wide variety of future research on network programmability.

# 9. REFERENCES


[1] R. K. Ahuja et al. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[2] C. J. Anderson et al. NetKAT: Semantic Foundations for Networks. In *POPL*, pages 113–126, Jan. 2014.

[3] Automatic test packet generation. https://github.com/eastzone/atpg.

[4] H. Ballani et al. Towards Predictable Datacenter Networks. In *SIGCOMM*, pages 242–253, Aug. 2011.

[5] C. Barnhart et al. Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *OR*, 48(2):318–326, Mar. 2000.

[6] A. Z. Broder et al. Static and Dynamic Path Selection on Expander Graphs: A Random Walk Approach. In *STOC*, pages 531–539, May 1997.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.

[8] A. Chakrabarti et al. Approximation Algorithms for the Unsplittable Flow Problem. In *APPROX*, pages 51–66, Sept. 2002.

[9] M. Chiosi et al. Network functions virtualisation introductory white paper. Technical report, ETSI, Oct. 2012.

[10] S. Chong et al. Secure Web Applications via Automatic Partitioning. In *SOSP*, pages 31–44, Oct. 2007.

[11] J. Chuzhoy and S. Li. A Polylogarithmic Approximation Algorithm for Edge-Disjoint Paths with Congestion 2. In *FOCS*, pages 233–242, Oct. 2012.

[12] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, Dec. 2004.

[14] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, Oct. 2007.

[15] Y. Dinitz et al. On the Single-Source Unsplittable Flow Problem. *Combinatorica*, 19(1):17–41, Jan. 1999.

[16] C. Dixon et al. ETTM: A Scalable Fault Tolerant Network Manager. In *NSDI*, pages 7–21, Mar. 2011.

[17] S. K. Fayazbakhsh et al. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 533–546, Berkeley, CA, USA, 2014. USENIX Association.

[18] A. Ferguson et al. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, pages 327–338, Aug. 2013.

[19] N. Foster et al. Frenetic: A Network Programming Language. In *ICFP*, pages 279–291, Sept. 2011.

[20] N. Foster, A. Guha, et al. The Frenetic Network Controller. In *The OCaml Users and Developers Workshop*, Sept. 2013.

[21] A. M. Frieze. Disjoint Paths in Expander Graphs via Random Walks: A Short Survey. In *RANDOM*, pages 1–14, Oct. 1998.

[22] A. Gember et al. Toward Software-Defined Middlebox Networking. In *HotNets*, pages 7–12, Oct. 2012.

[23] L. Glendenning et al. Scalable Consistency in Scatter. In *SOSP*, pages 15–28, 2011.

[24] P. B. Godfrey et al. Pathlet Routing. *SIGCOMM CCR*, 39(4):111–122, Aug. 2009.

[25] Gurobi Optimization Inc. The Gurobi optimizer. Available at http://www.gurobi.com.

[26] T. Hinrichs et al. Practical Declarative Network Management. In *WREN*, pages 1–10, 2009.

[27] P. Hooimeijer. Dprle decision procedure library.

[28] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[29] The internet topology zoo. http://www.topology-zoo.org.

[30] S. Ioannidis et al. Implementing a Distributed Firewall. In *CCS*, pages 190–199, Nov. 2000.

[31] S. Jain et al. B4: Experience with a Globally Deployed Software Defined WAN. In *SIGCOMM*, pages 3–14, Aug. 2013.

[32] M. Jelasity et al. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, Jan. 2009.

[33] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, pages 297–312, Apr. 2013.

[34] D. A. Joseph et al. A Policy-aware Switching Layer for Data Centers. In *SIGCOMM*, pages 51–62, Aug. 2008.

[35] N. Kang et al. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *9th CoNext*, pages 13–24, Dec. 2013.

[36] J. Kleinberg and R. Rubinfeld. Short paths in expander graphs. In *FOCS*, pages 86–95, Oct. 1996.

[37] J. M. Kleinberg. Single-Source Unsplittable Flow. In *FOCS*, pages 68–77, Oct. 1996.

[38] E. Kohler et al. The Click Modular Router. *TOCS*, 18(3):263–297, Aug. 2000.

[39] S. G. Kolliopoulos and C. Stein. Approximation Algorithms for Single-Source Unsplittable Flow. *SIAM J. Comput.*, 31(3):919–946, June 2001.

[40] T. Koponen et al. Network virtualization in multi-tenant datacenters. pages 203–216, Apr. 2014.

[41] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[42] L. Lamport. The part-time parliament. *TOCS*, 16(2):133–169, May 1998.

[43] J. Liu et al. Fabric: A Platform for Secure Sistributed Computation and Storage. In *SIGOPS EW*, pages 321–334, Oct. 2009.

[44] P. Marandi et al. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN*, pages 527 –536, May 2010.

[45] N. McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, Mar. 2008.

[46] C. Monsanto et al. A Compiler and Run-time System for Network Programming Languages. In *POPL*, pages 217–230, Jan. 2012.

[47] C. Monsanto et al. Composing Software-Defined Networks. In *NSDI*, pages 1–13, Apr. 2013.

[48] L. D. Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[49] G. C. Necula. Proof-Carrying Code. In *POPL*, pages 106–119, Jan. 1997.

[50] T. Nelson et al. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, Apr. 2014.

[51] H. Okamura and P. D. Seymour. Multicommodity Flows in Planar Graphs. *J. Combinatorial Theory Ser. B*, 31(1):75–81, 1981.

[52] L. Popa et al. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, pages 187–198, Aug. 2012.

[53] Puppet. http://puppetlabs.com.

[54] Z. A. Qazi et al. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, pages 27–38, Aug. 2013.

[55] M. Roesch. Snort—Lightweight Intrusion Detection for Networks. In *LISA*, pages 229–238, Nov. 1999.

[56] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR*, 22(4):299–319, Dec. 1990.

[57] V. Sekar et al. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, pages 24–38, Apr. 2012.

[58] J. Sherry et al. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, pages 13–24, Aug. 2012.

[59] A. Shieh et al. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, pages 1–8, June 2010.

[60] E. G. Sirer et al. Logical Attestation: An Authorization Architecture for Trustworthy Computing. In *SOSP*, pages 249–264, Oct. 2011.

[61] R. Soulé et al. Managing the Network with Merlin. In *HotNets*, Nov. 2013.

[62] R. Subramaniyan et al. GEMS: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. *Cluster Computing*, 9(1):101–120, Jan. 2006.

[63] R. Van Renesse et al. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *TOCS*, 21(2):164–206, Feb. 2003.

[64] A. Voellmy et al. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, pages 87–98, Aug. 2013.