Maple: Simplifying SDN Programming Using Algorithmic Policies

Andreas Voellmy* Junchang Wang*† Y. Richard Yang* Bryan Ford* Paul Hudak*
*Yale University †University of Science and Technology of China
{andreas.voellmy, junchang.wang, yang.r.yang, bryan.ford, paul.hudak}@yale.edu

ABSTRACT

Software-Defined Networking offers the appeal of a simple, centralized programming model for managing complex networks. However, challenges in managing low-level details, such as setting up and maintaining correct and efficient forwarding tables on distributed switches, often compromise this conceptual simplicity. In this paper, we present Maple, a system that simplifies SDN programming by (1) allowing a programmer to use a standard programming language to design an arbitrary, *centralized* algorithm, which we call an *algorithmic policy*, to decide the behaviors of an entire network, and (2) providing an abstraction that the programmer-defined, centralized policy runs, conceptually, "afresh" on every packet entering a network, and hence is oblivious to the challenge of translating a high-level policy into sets of rules on distributed individual switches. To implement algorithmic policies efficiently, Maple includes not only a highly-efficient multicore scheduler that can scale efficiently to controllers with 40+ cores, but more importantly a novel *tracing* runtime optimizer that can automatically record reusable policy decisions, offload work to switches when possible, and keep switch flow tables up-to-date by dynamically tracing the dependency of policy decisions on packet contents as well as the *environment* (system state). Evaluations using real HP switches show that Maple optimizer reduces HTTP connection time by a factor of 100 at high load. During simulated benchmarking, Maple scheduler, when not running the optimizer, achieves a throughput of over 20 million new flow requests per second on a single machine, with 95-percentile latency under 10 ms.

Categories and Subject Descriptors: C.2.3 [**Computer Communication Networks**]: Network Operations—*Network management*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Incremental compilers, Run-time environments, Optimization*.
General Terms: Algorithms, Design, Languages, Performance.
Keywords: Software-defined Networking, Policies, Openflow.

1. INTRODUCTION

A major recent development in computer networking is the notion of *Software-Defined Networking* (SDN), which allows a network to customize its behaviors through centralized policies at a conceptually centralized network controller. In particular, Openflow [13] has made significant progress by establishing (1) flow tables as a standard data-plane abstraction for distributed switches, (2) a protocol for the centralized controller to install flow rules and query states at switches, and (3) a protocol for a switch to forward to the controller packets not matching any rules in its switch-local flow table. These contributions provide critical components for realizing the vision that an operator configures a network by writing a simple, centralized network control program with a global view of network state, decoupling network control from the complexities of managing distributed state. We refer to the programming of the centralized controller as *SDN programming*, and a network operator who conducts SDN programming as an *SDN programmer*, or just programmer.

Despite Openflow's progress, a major remaining component in realizing SDN's full benefits is the SDN programming framework: the programming language and programming abstractions. Existing solutions require either explicit or restricted declarative specification of flow patterns, introducing a major source of complexity in SDN programming. For example, SDN programming using NOX [8] requires that a programmer explicitly create and manage flow rule patterns and priorities. Frenetic [7] introduces higher-level abstractions but requires restricted declarative queries and policies as a means for introducing switch-local flow rules. However, as new use cases of SDN continue to be proposed and developed, a restrictive programming framework forces the programmer to think within the framework's - rather than the algorithm's - structure, leading to errors, redundancy and/or inefficiency.

This paper explores an SDN programming model in which the programmer defines network-wide forwarding behaviors with the application of a high-level algorithm. The programmer simply defines a function f, expressed in a general-purpose programming language, which the centralized controller *conceptually* runs on every packet entering the network. In creating the function f, the programmer does not need to adapt to a new programming model but uses standard languages to design arbitrary algorithms for forwarding input packets. We refer to this model as *SDN programming of algorithmic policies* and emphasize that algorithmic policies and declarative policies do not exclude each other. Our system supports both, but this paper focuses on algorithmic policies.

The promise of algorithmic policies is a simple and flexible conceptual model, but this simplicity may introduce performance bottlenecks if naively implemented. Conceptually, in this model, the function f is invoked on every packet, leading to a serious computational bottleneck at the controller; that is, the controller may not have sufficient computational capacity to invoke f on every packet. Also, even if the controller's computational capacity can scale, the bandwidth demand that every packet go through the controller may be impractical. These bottlenecks are in addition to the extra latency of forwarding all packets to the controller for processing [6].

Rather than giving up the simplicity, flexibility, and expressive power of high-level programming, we introduce Maple, an SDN programming system that addresses these performance challenges. As a result, SDN programmers enjoy simple, intuitive SDN programming, while achieving high performance and scalability.

Specifically, Maple introduces two novel components to make SDN programming with algorithmic policies scalable. First, it introduces a novel SDN *optimizer* that "discovers" reusable forwarding decisions from a generic running control program. Specifically, the optimizer develops a data structure called a *trace tree* that records the invocation of the programmer-supplied f on a specific packet, and then generalizes the dependencies and outcome to *other* packets. As an example, f may read only one specific field of a packet, implying that the policy's output will be the same for any packet with the same value for this field. A trace tree captures the reusability of previous computations and hence substantially reduces the number of invocations of f and, in turn, the computational demand, especially when f is expensive.

The construction of trace trees also transforms arbitrary algorithms to a normal form (essentially a cached data structure), which allows the optimizer to achieve *policy distribution*: the generation and distribution of switch-local forwarding rules, totally transparently to the SDN programmer. By pushing computation to distributed switches, Maple significantly reduces the load on the controller as well as the latency. Its simple, novel translation and distribution technique optimizes individual switch flow table resource usage. Additionally, it considers the overhead in updating flow tables and takes advantage of multiple switch-local tables to optimize network-wide forwarding resource usage.

Maple also introduces a scalable run-time *scheduler* to complement the optimizer. When flow patterns are inherently non-localized, the central controller will need to invoke f many times, leading to scalability challenges. Maple's scheduler provides substantial horizontal scalability by using multi-cores.

We prove the correctness of our key techniques, describe a complete implementation of Maple, and evaluate Maple through benchmarks. For example, using HP switches, Maple optimizer reduces HTTP connection time by a factor of 100 at high load. Maple's scheduler can scale with 40+ cores, achieving a simulated throughput of over 20 million requests/sec on a single machine.

The aforementioned techniques have limitations, and hence programmers may write un-scalable f. Worst-case scenarios are that the computation of f is (1) not reusable (*e.g.*, depending on packet content), or (2) difficult to parallelize (*e.g.*, using shared states). Maple cannot make every controller scalable, and Maple programmers may need to adjust their designs or goals for scalability.

The rest of the paper is organized as follows. Section 2 motivates Maple using an example. Section 3 gives an overview of Maple architecture. Sections 4 and 5 present the details of the optimizer and scheduler, respectively. We present evaluations in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. A MOTIVATING EXAMPLE

To motivate algorithmic policies, consider a network whose policy consists of two parts. First, a secure routing policy: TCP flows with port 22 use secure paths; otherwise, the default shortest paths are used. Second, a location management policy: the network updates the location (arrival port on ingress switch) of each host. Specifying the secure routing policy requires algorithmic programming beyond simple GUI configurations, because the secure paths are computed using a customized routing algorithm.

To specify the preceding policy using algorithmic policies, an SDN programmer defines a function `f` to be invoked on every packet `pkt` arriving at `pkt.inport()` of switch `pkt.switch()`:

```
def f(pkt):
  srcSw = pkt.switch(); srcInp = pkt.inport()
  if locTable[pkt.eth_src()] != (srcSw,srcInp):
    invalidateHost(pkt.eth_src())
    locTable[pkt.eth_src()] = (srcSw,srcInp)
  dstSw = lookupSwitch(pkt.eth_dst())
  if pkt.tcp_dst_port() == 22:
    outcome.path = securePath(srcSw,dstSw)
  else:
    outcome.path=shortestPath(srcSw,dstSw)
  return outcome
```

The function `f` is simple and intuitive. The programmer does not think about or introduce forwarding rules—it is the responsibility of the programming framework to derive those automatically.

Unfortunately, current mainstream SDN programming models force programmers to explicitly manage low-level forwarding rules. Here is a controller using current programming models:

```
def packet_in(pkt):
  srcSw = pkt.switch(); srcInp = pkt.inport()
  locTable[pkt.eth_src()] = (srcSw,srcInp)
  dstSw = lookupSwitch(pkt.eth_dst())
  if pkt.tcp_dst_port() == 22:
   (nextHop,path)=securePath(srcSw,dstSw)
  else:
   (nextHop,path)=shortestPath(srcSw,dstSw)
  fixEndPorts(path,srcSw,srcInp,pkt.eth_dst())
  for each sw in path:
   inport' = path.inportAt(sw)
   outport'= path.outportAt(sw)
   installRule(sw,inport',
               exactMatch(pkt),
               [output(outport')])
  forward(srcSw,srcInp,action,nextHop)
```

We see that the last part of this program explicitly constructs and installs rules for each switch. The program uses the `exactMatch` function to build a match condition that includes all L2, L3, and L4 headers of the packet, and specifies the switch and incoming port that the rule pertains to, as well as the action to forward on port `outport'`. Thus, the program must handle the complexity of creating and installing forwarding rules.

Unfortunately, the preceding program may perform poorly, because it uses only exact match rules, which match at a very granular level. Every new flow will fail to match at switch flow tables, leading to a controller RTT. To make matters worse, it may not be possible to completely cover active flows with granular rules, if rule space in switches is limited. As a result, rules may need to be frequently evicted to make room for new rules.

Now, assume that the programmer realizes the issue and decides to conduct an optimization to use *wildcard* rules. She might replace the exact match rule with the following:

```
installRule(sw,inport',
          {from:pkt.eth_src(), to:pkt.eth_dst()},
          [output(outport')])
```

Unfortunately, this program has bugs! First, if a host *A* sends a TCP packet that is not to port 22 to another host *B*, then rules with wildcard will be installed for *A* to *B* along the shortest path. If *A* later initiates a new TCP connection to port 22 of host *B*, since the packets of the new connection match the rules already installed, they will not be sent to the controller and as a result, will not be sent along the desired path. Second, if initially host *A* sends a packet to port 22 of *B*, then packets to other ports will be misrouted.

To fix the bugs, the programmer must prioritize rules with appropriate match conditions. A program fixing these bugs is:

```
def packet_in(pkt):
 ...
 if pkt.tcp_dst_port() == 22:
   (nextHop,path)=securePath(srcSw,dstSw)
   fixEndPorts(path,srcSw,srcInp,pkt.eth_dst())
   for each sw in path:
     inport' = path.inportAt(sw)
     outport'= path.outportAt(sw)
     installRule(sw,inport',priority=HIGH,
                {from:pkt.eth_src(),
                 to:pkt.eth_dst(),to_tcp_port:22},
                [output(output')])
   forward(srcSw,srcInp,action,nextHop)
 else:
   (nexthop,path)=shortestPath(srcSw,dstSw)
   fixEndPorts(path,srcSw,srcInp,pkt.eth_dst())
   for each sw in path:
    inport' = path.inportAt(sw)
    outport'= path.outportAt(sw)
    installRule(sw,inport',priority=MEDIUM,
               {from:pkt.eth_src(),
                to:pkt.eth_dst(),to_tcp_port:22},
               [output(toController)])
   installRule(sw,inport',priority=LOW,
         {from:pkt.eth_src(),to:pkt.eth_dst()},
         [output(outport')])
  forward(srcSw,srcInp,action,nextHop)
```

This program is considerably more complex. Consider the `else` statement. Since it handles non port 22 traffic, it installs wildcard rules at switches along the shortest path, not the secure path. Although the rules are intended for only non port 22 traffic, since flow table rules do not support negation (*i.e.*, specifying a condition of port != 22), it installs wildcard rules that match all traffic from src to dst. To avoid such a wildcard rule being used by port 22 traffic, it adds a special rule, whose priority (`MEDIUM`) is higher than that of the wildcard rule, to prevent the wildcard from being applied to port 22 traffic. This is called a barrier rule. Furthermore, the program may still use resources inefficiently. For example, for some host pairs, the most secure route may be identical to the shortest path. In this case, the program will use more rules than necessary.

Comparing the example programs using the current models with the function `f` defined at the beginning of this section, we see the unnecessary burden that current models place on programmers, forcing them to consider issues such as match granularity, encoding of negation using priorities, and rule dependencies. One might assume that the recent development of declarative SDN programming, such as using a data query language, may help. But such approaches require that a programmer extract decision conditions (*e.g.*, conditional and loop conditions) from an algorithm and express them declaratively. This may lead to easier composition and construction of flow tables, but it still places the burden on the programmer, leading to errors, restrictions, and/or redundancies.

3. ARCHITECTURE OVERVIEW

The core objective of Maple is to offer an SDN programmer the abstraction that a general-purpose program f defined by the programmer runs "from scratch" on a centralized controller for every packet entering the network, hence removing low-level details, such as distributed switch flow tables, from the programmer's conceptual model. A naive implementation of this abstraction, however, would of course yield unusable performance and scalability.

Maple introduces two key components in its design to efficiently implement the abstraction. The first is an optimizer, or tracing runtime, which automatically discovers reusable (cachable) algorithmic policy executions at runtime, offloads work to switches when possible, and invalidates cached policy executions due to environment changes. One can build a proactive rule installer on top of the tracing runtime, for example, using historical packets, or develop a static analyzer to proactively evaluate f. These are out of the scope of this paper. The second is a run-time scheduler, or scheduler for short, which provides Maple scalable execution of policy "misses" generated by the many switches in a large network on multicore hardware. Figure 1 illustrates the positions of the two components.

In addition, Maple allows a set of higher-level tools to be built on top of the basic abstraction: (1) deployment portal, which imposes constraints on top of Maple, in forms such as best practices, domain-specific analyzers, and/or higher-level, limited configuration interfaces to remove some flexibility of Maple; and (2) policy composer, which an SDN programmer can introduce.

This paper focuses on the basic abstraction, the optimizer, and the scheduler. This section sketches a high-level overview of these three pieces, leaving technical details to subsequent sections.

3.1 Algorithmic Policy `f`

An SDN programmer specifies the routing of each packet by providing a function `f`, conceptually invoked by our run-time scheduler on each packet. An SDN programmer can also provide handlers for other events such as switch-related events, but we focus here on packet-arrival events. Although the function `f` may in principle be expressed in any language, for concreteness we illustrate `f` in a functional style:

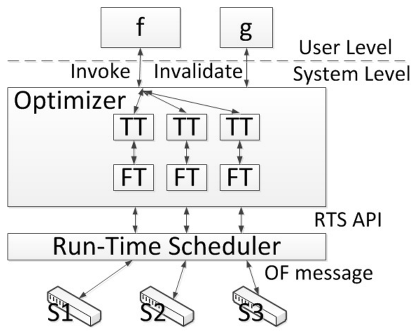


Figure 1: Maple system components.

```
f :: (PacketIn,Env) -> ForwardingPath
```

Specifically, `f` is given two inputs: `PacketIn`, consisting of a packet (`Packet`) and the ingress switch/port; and `Env`, a handle to an environment context. The objective of (`Env`) is to provide `f` with access to Maple-maintained data structures, such as a network information base that contains the current network topology.

The return value of a policy `f` is a forwarding path, which specifies whether the packet should be forwarded at all and if so how. For multicast, this result may be a tree instead of a linear path. The return of `f` specifies global forwarding behavior through the network, rather than hop-by-hop behavior.

Except that it must conform to the signature, `f` may use arbitrary algorithms to classify the packets (*e.g.*, conditional and loop statements) and compute forwarding actions (*e.g.*, graph algorithms).

3.2 Optimizer

Although a policy `f` might in principle follow a different execution path and yield a different result for every packet, in practice many packets—and often many flows—follow the same or similar execution paths in realistic policies. For example, consider the example algorithmic policy in Section 2. `f` assigns the same path to two packets if they match on source and destination MAC addresses and neither has a TCP port value 22. Hence, if we invoke `f` on one packet, and then a second packet arrives, and the two packets satisfy the preceding condition, then the first invocation of `f` is reusable for the second packet. The key objective of the optimizer is to leverage these reusable algorithm executions.

3.2.1 *Recording reusable executions*

The technique used by Maple to detect and utilize reusable executions of a potentially complex program `f` is to record the essence of its decision dependencies: the data accesses (*e.g.*, reads and assertions) of the program on related inputs. We call a sequence of such data accesses a *trace*, and Maple obtains traces by logging data accesses made by `f`.

As an example, assume that the log during one execution of an f is follows: (1) the only data access of the program is to apply a test on TCP destination port for value 22, (2) the test is true, and (3) the program drops the packet. One can then infer that if the program is again given an arbitrary packet with TCP destination port 22, the program will similarly choose to drop the packet.

The key data structure maintained by the optimizer is a collection of such data access traces represented as a *trace tree*. Figure 2 is a trace tree from 3 traces of a program `f`. In one of these traces, the program first tests TCP destination port for 22 and the result is false (the right branch is false). The program then reads the field value of Ethernet destination (=4) and Ethernet source (=6), resulting in

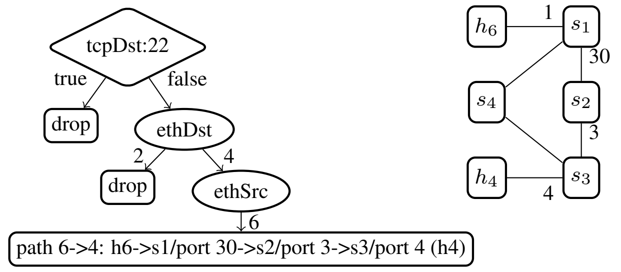


Figure 2: An example trace tree. Diamond indicates a test of some condition, circle indicates reading an attribute, and rectangles contain return values. tcpDst denotes TCP destination; ethDst and ethSrc denote Ethernet destination and source. Topology of the network is shown on the right.

the program's decision to forward packets from host 6 to host 4 along the shortest path between these two hosts. For concreteness, Figure 2 shows the details of how a path is represented. Assume that host 6 is attached at switch s1/port 1, and host 4 at switch s3/port 4. Figure 2 shows the detail of the path from host 6 to host 4 as: from host 6 (s1/port 1) to output port 30 of s1, which is connected to switch s2; s2 sends out at port 3, which is connected to s3, where host 4 is attached at port 4. The trace tree abstracts away the details of `f` but still retains its output decisions as well as the decisions' dependencies on the input packets.

3.2.2 Utilizing distributed flow tables

Merely caching prior policy decisions using trace trees would not make SDN scalable if the controller still had to apply these decisions centrally to every packet. Real scalability requires that the controller be able to "push" many of these packet-level decisions out into the flow tables distributed on the individual Openflow switches to make quick, on-the-spot per-packet decisions.

To achieve this goal, the optimizer maintains, logically, a trace tree for each switch, so that the leaves for a switch's trace tree contain the forwarding actions required for that switch only. For example, for the trace tree shown in Figure 2, the switch-specific trace tree maintained by Maple for switch s1 has the same structure, but includes only port actions for switch s1 at the leaves (*e.g.*, the right most leave is labeled only port 30, instead of the whole path).

Given the trace tree for a switch, the optimizer compiles the trace tree to a prioritized set of flow rules, to form the flow table of the switch. In particular, there are two key challenges to compile an efficient flow table for a switch. First, the table size at a switch can be limited, and hence it is important to produce a compact table to fit more cached policy decisions at the switch. Second, the optimizer will typically operate in an online mode, in which it needs to continuously update the flow table as new decisions are cached. Hence, it is important to achieve fast, efficient flow table updates. To address the challenges, our optimizer introduces multiple techniques: (1) it uses incremental compilation, avoiding full-table compilation; (2) it optimizes the number of rules used in a flow table, through both switch-local and network-wide optimizations on switch tables; and (3) it minimizes the number of priorities used in a flow table, given that the update time to a flow table is typically proportional to the number of priority levels [18].

3.2.3 Keeping trace trees, flow tables up-to-date

Just as important as using distributed flow tables efficiently is keeping them up-to-date, so that stale policy decisions are not applied to packets. Specifically, the decision of `f` on a packet depends on not only the fields of the packet, but also other variables. For example, data accesses by `f` through the `Env` handle to access the network information base will also generate dependency, which Maple tracks. Hence, trace trees record the dependencies of prior policy decisions on not only packet fields but also Maple-maintained environment state such as network topology and configurations.

A change to the environment state may invalidate some part of the trace trees, which in turn may invalidate some entries of the flow tables. There is a large design spectrum on designing the invalidation scheme, due to various trade-offs involving simplicity, performance, and consistency. For example, it can be simple and "safe" to invalidate more flow table entries than necessary—though doing so may impact performance. To allow extensible invalidation design, Maple provides a simple invalidation/update API, in which a flexible selection clause can be specified to indicate the cached computations that will be invalidated. Both system event handlers and user-defined functions (*i.e.*, `g` in Figure 1) can issue these API calls. Hence, user-defined `f` can introduce its own (persistent) environment state, and manage its consistency. Note that dependencies that Maple cannot automatically track will require user-initiated invalidations (see Section 4.2) to achieve correctness.

3.3 Multicore Scheduler

Even with efficient distributed flow table management, some fraction of the packets will "miss" the cached policy decisions at switches and hence require interaction with the central controller. This controller-side processing of misses must scale gracefully if the SDN as a whole is to scale.

Maple therefore uses various techniques to optimize the controller's scalability, especially for current and future multicore hardware. A key design principle instrumental in achieving controller scalability is *switch-level parallelism*: designing the controller's thread model, memory management, and event processing loops to localize controller state relevant to a particular "client" switch. This effectively reduces the amount and frequency of accesses to state shared across the processing paths for multiple client switches.

While many of our design techniques represent "common sense" and are well-known in the broader context of parallel/multicore software design, all of the SDN controllers that we had access to showed major scalability shortcomings, as we explore later in Section 6. We were able to address most of these shortcomings through a judicious application of switch-level parallelism principles, such as buffering and batching input and output message streams, and appropriate scheduling and load balancing across cores. Section 5 discusses scheduling considerations further.

4. MAPLE OPTIMIZER

This section details the optimizer, highlighting the construction and invalidation of trace trees and methods for converting trace trees to flow tables. We choose to present out ideas in steps, from basic ideas to optimizations, to make understanding easier.

4.1 Basic Concepts

Trace tree: A trace tree provides an abstract, partial representation of an algorithmic policy. We consider packet attributes $a_1, \dots a_n$ and write $p.a$ for the value of the a attribute of packet p. We write $dom(a)$ for the set of possible values for attribute a: $p.a \in dom(a)$ for any packet p and attribute a.

DEFINITION 1 (TRACE TREE). *A* trace tree (TT) *is a rooted tree where each node t has a field type$_t$ whose value is one of* $\mathbf{L}$ *(leaf),* $\mathbf{V}$ *(value),* $\mathbf{T}$ *(test), or* $\mathbf{\Omega}$ *(empty) and such that:*

1. *If type$_t$ $= \mathbf{L}$, then t has a $value_t$ field, which ranges over possible return values of the algorithmic policy. This node represents the behavior of a program that returns $value_t$ without inspecting the packet further.*

2. *If type$_t$ $= \mathbf{V}$, then t has an $attr_t$ field, and a $subtree_t$ field, where $subtree_t$ is an associative array such that $subtree_t[v]$*

Algorithm 1: SEARCHTT(t, p)

```
while true do
    if $type_t = \mathbf{\Omega}$ then
        return NIL;
    else if $type_t = \mathbf{L}$ then
        return $value_t$;
    else if $type_t = \mathbf{V} \wedge p.attr_t \in keys(subtree_t)$ then
        $t \leftarrow subtree_t[p.attr_t]$;
    else if $type_t = \mathbf{V} \wedge p.attr_t \notin keys(subtree_t)$ then
        return NIL;
    else if $type_t = \mathbf{T} \wedge p.attr_t = value_t$ then
        $t \leftarrow t_+$;
    else if $type_t = \mathbf{T} \wedge p.attr_t \neq value_t$ then
        $t \leftarrow t_-$;
```

is a trace tree for value $v \in keys(subtree_t)$. This node represents the behavior of a program that if the supplied packet p satisfies $p.attr_t = v$, then it continues to $subtree_t[v]$.

3. *If $type_t = \mathbf{T}$, then t has an $attr_t$ field, a $value_t$ field, such that $value_t \in dom(attr_t)$, and two subtree fields t_+ and t_-. This node reflects the behavior of a program that tests the assertion $p.attr_t = value_t$ of a supplied packet p and then branches to t_+ if true, and t_- otherwise.*

4. *If $type_t = \mathbf{\Omega}$, then t has no fields. This node represents arbitrary behavior (*i.e.*, an unknown result).*

Given a TT, one can look up the return value of a given packet, or discover that the TT does not include a return value for the packet. Algorithm 1 shows the SEARCHTT algorithm, which defines the semantics of a TT. Given a packet and a TT, the algorithm traverses the tree, according to the content of the given packet, terminating at an $\mathbf{L}$ node with a return value or an $\mathbf{\Omega}$ node which returns NIL.

Flow table (FT): A pleasant result is that given a trace tree, one can generate an Openflow flow table (FT) efficiently.

To demonstrate this, we first model an FT as a collection of *FT rules*, where each FT rule is a triple $(priority, match, action)$, where $priority$ is a natural number denoting its priority, with the larger the value, the higher the priority; $match$ is a collection of zero or more (packet attribute, value) pairs, and $action$ denotes the forwarding action, such as a list of output ports, or ToController, which denotes sending to the controller. Matching a packet in an FT is to find the highest priority rule whose $match$ field matches the packet. If no rule is found, the result is ToController. Note that FT matches do not support negations; instead, priority ordering may be used to encode negations (see below).

Trace tree to forwarding table: Now, we describe BUILDFT(t), a simple algorithm shown in Algorithm 2 that compiles a TT rooted at node t into an FT by recursively traversing the TT. It is simple because its algorithm structure is quite similar to the standard in-order tree traversal algorithm. In other words, the elegance of the TT representation is that one can generate an FT from a TT using basically simple in-order tree traversal.

Specifically, BUILDFT(t), which starts at line 1, first initializes the global $priority$ variable to 0, and then starts the recursive BUILD procedure. Note that each invocation of BUILD is provided with not only the current TT node t, but also a match parameter denoted m, whose function is to accumulate attributes read or positively tested along the path from the root to the current node t. We can see that BUILDFT(t) starts BUILD with m being any (*i.e.*, match-all-packets). Another important variable maintained by BUILD is $priority$. One can observe an invariant (lines 8 and 16)

Algorithm 2: BUILDFT(t)

```
1  Algorithm BUILDFT ($t$)
2      $priority \leftarrow 0$;
3      BUILD($t$,$any$);
4      return;
5  Procedure BUILD ($t$,$m$)
6      if $type_t = L$ then
7          $emitRule(priority, m, value_t)$;
8          $priority \leftarrow priority + 1$;
9      else if $type_t = V$ then
10         for $v \in keys(subtrees_t)$ do
11             BUILD$(subtrees_t[v], m \wedge (attr_t : v))$;
12     else if $type_t = T$ then
13         BUILD$(t_-, m)$;
14         $m_t = m \wedge (attr_t : value_t)$;
15         $emitRule(priority, m_t,$ ToController$)$;
16         $priority \leftarrow priority + 1$;
17         BUILD$(t_+, m_t)$;
```

that its value is increased by one after each output of an FT rule. In other words, BUILD assigns each FT rule with a priority identical to the order in which the rule is added to the FT.

BUILD processes a node t according to its type. First, at a leaf node, BUILD emits an FT rule (at line 7) with the accumulated match m and the designated action at the leaf node. The priority of the FT rule is the current value of the global variable $priority$. Second, at a $\mathbf{V}$ node (line 9), BUILD recursively emits the FT rules for each subtree branch. Before proceeding to branch with value v, BUILD adds the condition on the branch (denoted $attr_t : value$) to the current accumulated match m. We write $m_1 \wedge m_2$ to denote the intersection of two matches.

The third case is a $\mathbf{T}$ node t. One might think that this is similar to a $\mathbf{V}$ node, except that a $\mathbf{T}$ node has only two branches. Unfortunately, flow tables do not support negation, and hence BUILD cannot include the negation condition in the accumulated match condition when recursing to the negative branch. To address the issue, BUILD uses the following techniques. (1) It emits an FT rule, which we call the *barrier* rule for the t node, with action being ToController and match m_t being the intersection of the assertion of the $\mathbf{T}$ node (denoted as $attr_t : value_t$) and the accumulated match m. (2) It ensures that the barrier rule has a higher priority than any rules emitted from the negative branch. In other words, the barrier rule prevents rules in the negated branch from being executed when m_t holds. (3) To avoid that the barrier rule blocks rules generated from the positive branch, the barrier rule should have lower priority than those from the positive branch. Formally, denote r_b as the barrier rule at a $\mathbf{T}$ node, r_- a rule from the negative branch, and r_+ a rule from the positive branch. BUILD needs to enforce the following ordering constraints:

$$r_- \rightarrow r_b \rightarrow r_+, \tag{1}$$

where $r_1 \rightarrow r_2$ means that r_1 has lower priority than r_2.

Since BUILD increases priority after each rule, enforcing the preceding constraints is easy: in-order traversal, first negative and then positive. One can verify that the run-time complexity of BUILDFT(t) is $\mathcal{O}(n)$, where n is the size of the trace tree rooted at t.

Example: We apply BUILDFT to the root of the trace tree shown in Figure 2, for switch s1 (*e.g.*, leaves containing only actions pertaining to s1, such as drop and port 30). Since the root of the tree is a $\mathbf{T}$ node, testing on TCP destination port 22, BUILD, invoked by BUILDFT, goes to line 12. Line 13 recursively calls BUILD on the negative (right) branch with match m still being any. Since the node (labeled ethDst) is a $\mathbf{V}$ node on the Ethernet destination attribute, BUILD proceeds to visit each subtree, adding a condition to match on the Ethernet destination labeled on the edge to the subtree. Since the subtree on edge labeled 2 is a leaf, BUILD (executing at line 7) emits an FT rule with priority 0, with a match on Ethernet destination 2. The action is to drop the packet. BUILD then increments the priority, returns to the parent, and visits the subtree labeled 4, which generates an FT rule at priority level 1 and increments the priority. After returning from the subtree labeled 4, BUILD backtracks to the **T** node (tcpDst:22) and outputs a barrier rule with priority 2, with match being the assertion of the node: matching on TCP destination port 22. BUILD outputs the final FT rule for the positive subtree at priority 3, dropping packets to TCP destination port 22. The final FT for switch s1 is:

```
[ (3, tcp_dst_port=22          , drop),
  (2, tcp_dst_port=22          , toController),
  (1, eth_dst=4 && eth_src=6, port 30),
  (0, eth_dst=2                , drop)]
```

If one examines the FT carefully, one may observe some inefficiencies in it, which we will address in Section 4.3. A key at this point is that the generated FT is correct:

THEOREM 1 (FT CORRECTNESS). *tree and* BUILDFT(*tree*) *encode the same function on packets.*

4.2 Trace Tree Augmentation & Invalidation

With the preceding basic concepts, we now describe our tracing runtime system, to answer the following questions: (1) how does Maple transparently generate a trace tree from an arbitrary algorithmic policy? (2) how to invalidate outdated portions of a trace tree when network conditions change?

Maple packet access API: Maple builds trace trees with a simple requirement from algorithmic policies: they access the values of packet attributes and perform boolean assertions on packet attributes using the Maple packet access API:

```
readPacketField :: Field -> Value
testEqual       :: (Field,Value) -> Bool
ipSrcInPrefix   :: IPPrefix -> Bool
ipDstInPrefix   :: IPPrefix -> Bool
```

The APIs simplify programming and allow the tracing runtime to observe the sequence of data accesses and assertions made by a policy. A language specific version of Maple can introduce wrappers for these APIs. For example, `pkt.eth_src()` used in Section 2 is a wrapper invoking `readPacketField` on Ethernet source.

Trace: Each invocation of an algorithmic policy that uses the packet access API on a particular packet generates a *trace*, which consists of a sequence of *trace items*, where each trace item is either a *Test* item, which records an assertion being made and its outcome, or a *Read* item, which records the field being read and the read value. For example, if a program calls `testEqual(tcpDst, 22)` on a packet and the return is false, a *Test* item with assertion of TCP destination port being 22 and outcome being false is added to the trace. If the program next calls `readPacketField(ethDst)` and the value 2 is returned, a *Read* item with field being Ethernet destination and value being 2 will be appended to the trace. Assume that the program terminates with a returned action of drop, then drop will be set as the returned action of the trace, and the trace is ready to be added to the trace tree.

Augment trace tree with a trace: Each algorithmic policy starts with an empty trace tree, represented as Ω. After collecting a new trace, the optimizer *augments* the trace tree with the new trace. The AUGMENTTT(t, *trace*) algorithm, presented in Algorithm 3, adds a new trace *trace* to a trace tree rooted at node t. The algorithm walks the trace tree and the trace in lock step to find the location at which to extend the trace tree. It then extends the trace tree at the found location with the remaining part of the trace. The algorithm uses $head(trace)$ to read the first item of a trace, and $next(trace)$ to remove the head and return the rest. The algorithm uses a straightforward procedure TRACETOTREE($trace$), which we omit here, that turns a linear list into a trace tree.

Algorithm 3: AUGMENTTT($t, trace$)

```
1  if $type_t = \Omega$ then
2      $t \leftarrow$ TRACETOTREE($trace$);
3      return;
4  repeat
5      $item = head(trace)$; $trace \leftarrow next(trace)$;
6      if $type_t = \mathbf{T}$ then
7          if $item.outcome$ is true then
8              if $type_{t_+} = \Omega$ then
9                  $t_+ \leftarrow$ TRACETOTREE($trace$);
10                 return;
11             else
12                 $t \leftarrow t_+$ ;
13         else
14             if $type_{t_-} = \Omega$ then
15                 $t_- \leftarrow$ TRACETOTREE($trace$);
16                 return;
17             else
18                 $t \leftarrow t_-$ ;
19     else if $type_t = \mathbf{V}$ then
20         if $item.value \in keys(subtree_t)$ then
21             $t \leftarrow subtree_t[item.value]$;
22         else
23             $subtree_t[item.value] \leftarrow$ TRACETOTREE($trace$);
24             return;
25 until;
```

Example: Figure 3 illustrates the process of augmenting an initially empty tree. The second tree results from augmenting the first tree with trace $Test(tcpDst, 22; \text{False})$, $Read(ethDst; 2)$; action=drop. In this step, AUGMENTTT calls TRACETOTREE at the root. Note that TRACETOTREE always places an Ω node in an unexplored branch of a **T** node, such as the t_+ branch of the root of the second tree. The third tree is derived from augmenting the second tree with the trace $Test(tcpDst, 22; \text{False})$, $Read(ethDst; 4)$, $Read(ethSrc, 6)$; action=*port 30*. In this case, the extension is at a **V** node. Finally, the fourth tree is derived by augmenting the third tree with trace $Test(tcpDst, 22; \text{True})$; action=*drop*. This step fills in the positive branch of the root.

Correctness: The trace tree constructed by the preceding algorithm returns the same result as the original algorithmic policy, when there is a match. Formally, we have:

THEOREM 2 (TT CORRECTNESS). *Let t be the result of augmenting the empty tree with the traces formed by applying the algorithmic policy f to packets $pkt_1 \dots pkt_n$. Then t safely represents f in the sense that if* SEARCHTT(t, pkt) *is successful, then it has the same answer as $f(pkt)$.*

Optimization: trace compression: A trace may have redundancy. Specifically, although the number of distinct observations that a program `f` can make of the packet is finite, a program may repeatedly observe or test the same attribute (field) of the packet, for example during a loop. This can result in a large trace, increasing the cost of tracing. Furthermore, redundant trace nodes may increase the size of the trace tree and the number of rules generated.

Maple applies COMPRESSTRACE, Algorithm 4, to eliminate both read and test redundancy in a trace before applying the preceding augmentation algorithm. In particular, the algorithm tracks the subset of packets that may follow the current trace. When it encounters

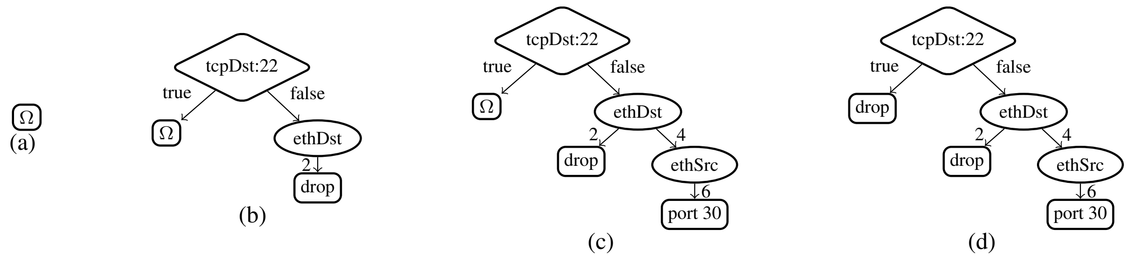


Figure 3: Augmenting a trace tree for switch s1. Trace tree starts as empty (Ω) as (a).

Algorithm 4: COMPRESSTRACE()

```
1 for next access entry on attribute $a$ do
2     if range(entry) included in knownRange then
3         ignore
4     else
5         update knownRange
```

a subsequent data access, it determines whether the outcome of this data access is completely determined by the current subset. If so, then the data access is ignored, since the program is equivalent to a similar program that simply omits this redundant check. Otherwise, the data access is recorded and the current subset is updated.

Maple trace tree invalidation API: In addition to the preceding packet access API, Maple also provides an API to invalidate part of a trace tree, where `SelectionClause` specifies the criteria:

```
invalidateIf :: SelectionClause -> Bool
```

For example, the `invalidateHost` call in the motivating example shown in Section 2 is a shorthand of invoking `invalidateIf` with `SelectionClause` as source or destination MAC addresses equal to a host's address. This simple call removes all outdated forwarding actions involving the host after it changes location.

As another example, if `SelectionClause` specifies a switch and port pair, then all paths using the switch/port pair (*i.e.*, link) will be invalidated. This topology invalidation is typically invoked by a Maple event listener that receives Openflow control events. Maple also contains an API to update the trace tree to implement fast-routing. This is out of scope of this paper.

4.3 Rule & Priority Optimization

Motivation: With the basic algorithms covered, we now present optimizations. We start by revisiting the example that illustrates BUILDFT at the end of Section 4.1. Let $\texttt{FT}_\texttt{B}$ denote the example FT generated. Consider the following FT, which we refer as $\texttt{FT}_\texttt{O}$:

```
[ (1, tcp_dst_port=22          , drop),
  (0, eth_dst==4 && eth_src==6, port 30),
  (0, eth_dst==2               , drop)]
```

One can verify that the two FTs produce the same result. In other words, the example shows that BUILDFT has two problems: (1) it may generate more flow table rules than necessary, since the barrier rule in $\texttt{FT}_\texttt{B}$ is unnecessary; and (2) it may use more priority levels than necessary, since $\texttt{FT}_\texttt{B}$ has 4 priorities, while $\texttt{FT}_\texttt{O}$ has only 2.

Reducing the number of rules generated for an FT is desirable, because rules are often implemented in TCAMs where rule space is limited. Reducing the number of priority levels is also beneficial, because TCAM update algorithms often have time complexity linear in the number of priority levels needed in a flow table [18].

Barrier elimination: We start with eliminating unnecessary barrier rules. BUILDFT outputs a barrier rule for each **T** node t. However, if the rules emitted from the positive branch t_+ of t is *complete* (*i.e.*, every packet matching t_+'s match condition is handled by t_+'s rules), then there is no need to generate a barrier rule for t, as the rules for t_+ already match all packets that the barrier rule would match. One can verify that checking this condition eliminates the extra barrier rule in $\texttt{FT}_\texttt{B}$.

Define a general predicate $isComplete(t)$ for an arbitrary tree node t: for an **L** node, $isComplete(t) = true$, since a BUILDFT derived compiler will generate a rule for the leaf to handle exactly the packets with the match condition; for a **V** node, $isComplete(t) = true$ if both $|subtrees_t| = |dom(attr_t)|$ and $isComplete(subtree_v)$ for each $v \in keys(subtrees_t)$; otherwise $isComplete(t) = false$ for the **V** node. For a **T** node t, $isComplete(t) = isComplete(t_-)$. We define $needsBarrier(t)$ at a **T** node t to be true if t_+ is not complete and t_- is not $\mathbf{\Omega}$, false otherwise.

Priority minimization: Minimizing the number of priorities is more involved. But as we will show, there is a simple, efficient algorithm achieving the goal, without the need to increase priority after outputting every single rule, as BUILDFT does.

Consider the following insight: since rules generated from different branches of a **V** node are disjoint, there is no need to use priority levels to distinguish them. Hence, the priority increment from 0 to 1 by BUILDFT for the example at the end of Section 4.1 is unnecessary. The preceding insight is a special case of the general insight: one can assign arbitrary ordering to two rules if their match conditions are disjoint.

Combining the preceding general insight with the ordering constraints shown in Equation (1) in Section 4.1, we define the minimal priority assignment problem as choosing a priority assignment P to rules with the minimal number of distinct priority values:

$$\begin{aligned} &\underset{P}{\text{minimize}} && |P| \\ &\text{subject to} && (r_i \to r_j) \wedge (r_i, r_j \text{ overlap}) : P(r_i) < P(r_j) \end{aligned} \tag{2}$$

One may solve Problem (2) using topological ordering. Specifically, construct a directed acyclic graph (DAG) $G_r = (V_r, E_r)$, where V_r is the set of rules, and E_r the set of ordering constraints among the rules: there is an edge from r_i to r_j iff $r_i \to r_j$ and r_i, r_j overlap. Initialize variable *priority* to 0. Assign rule priorities as follows: select all nodes without any incoming edges, assign them *priority*, delete all such nodes, increase *priority* by 1, and then repeat until the graph is empty.

An issue of the preceding algorithm is that it needs to first generate all rules, compute their ordering constraints, and then assign priorities. However, Maple's trace tree has special structures that allow us to make simple changes to BUILDFT to still use in-order trace tree traversal to assign minimal priority levels. The algorithm that we will design is simple, more efficient, and also better suited for incremental updates.

Define a weighted graph $G_o(V_o, E_o, W_o)$ for a trace tree to capture all ordering constraints. Specifically, V_o is the set of all trace tree nodes. All edges of the trace tree, except those from a **T** node t to t_-, belong to E_o as well. To motivate additional edges in E_o, consider all ordering constraints among the rules. Note that the edges of a trace tree do not include constraints of the form $r_- \to r_b$, as defined in Equation (1). To introduce such constraints in G_o, extend E_o to include *up-edges* from rule-generating nodes (*i.e.*, **T** nodes with barriers or **L** nodes). Identifying the up-edges from a rule-generating node n is straightforward. Consider the path from the tree root to node n. Consider each $\mathbf{T}$ node t along path. If n is on the negative branch of t and the intersection of the accumulated match conditions of t_+ and n is nonempty, then there should be an up-edge from n to t. One can verify that G_o is a DAG.

To remove the effects of edges that do not convey ordering constraints, we assign all edges weight 0 except the following:

1. For an edge from a $\mathbf{T}$ node t to t_+, this edge has $w(e) = 1$ if t needs a barrier and $w(e) = 0$ otherwise;
2. For each up-edge, the weight is 1.

Example: Figure 4 shows G_o for Figure 3(d). Dashed lines are up-edges of E_o. Edges that are in both the TT and E_o are shown as thick solid lines, while edges that are only in TT are shown as thin solid lines. E_o edges with weight 1 are labeled $w = 1$. The `drop` leaf in the middle has an up-edge to the root, for example, because its accumulated match condition is *ethDst* : 2, which overlaps the root's positive subtree match condition of *tcpDst* : 22. The E_o edge to the positive subtree of the root has weight 0, because the root node is complete and hence no barrier is needed for it.

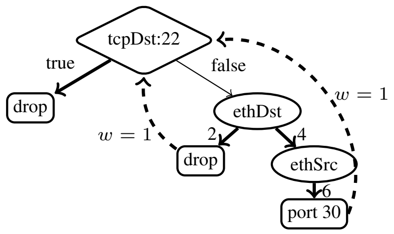


Figure 4: Order graph G_o for trace tree of Figure 3(d).

Algorithm: The algorithm based on G_o to eliminate barriers and minimize priorities is shown in OPTBUILDFT. The algorithm traverses the TT in-order, the same as BUILDFT. Each TT node t has a variable $priority(t)$, which is initialized to 0, and is incremented as the algorithm runs, in order to be at least equal to the priority of a processed node x plus the link weight from x to t. Since all non-zero weight edges flow from right to left (*i.e.*, negative to positive) in G_o, in-order traversal guarantees that when it is time to output a rule at a node, all nodes before it in the ordering graph have already been processed, and hence the node's priority is final.

Example 1: Consider executing OPTBUILDFT on the example of Figure 4. The algorithm begins by processing the negative branch of the root. At the *ethDst* $\mathbf{V}$ node, OPTBUILDFT updates the priority of its two children with priority 0, since its priority is 0 and the edge weights to its children are 0. The algorithm then processes the *drop* leaf for *ethDst* : 2, emits a rule at priority 0, and updates the root priority to 1, since its priority is 0 and its up-edge weight is 1. The algorithm then processes the *ethDst* : 4 branch where it processes the leaf in a similar way. After backtracking to the root, OPTBUILDFT skips the barrier rule, since its positive subtree is complete, and updates the priority of its positive child to be 1 (its priority), since the edge weight to the positive child is 0. Finally, it proceeds to the positive child and outputs a drop rule at priority 1.

Example 2: Consider the following policy that tests membership of the IP destination of a packet in a set of IP prefixes:

```
f(p):  if p.ipDstInPrefix(103.23.0.0/16):
         if p.ipDstInPrefix(103.23.3.0/24):
           return a
         else:
           return b
       if p.ipDstInPrefix(101.1.0.0/16):
         return c
       if p.ipDstInPrefix(101.0.0.0/13):
         return d
       return e
```

Algorithm 5: OPTBUILDFT(t)

```
Algorithm OPTBUILDFT ($t$)
    OPTBUILD($t$, $any$);
    return;
Procedure update ($t$)
    for $(t, x) \in E_o$ do
        $priority(x) =$
        $\max(priority(x), weight(t, x) + priority(t))$;
Procedure OPTBUILD ($t$,$m$)
    if $type_t = L$ then
        $emitRule(priority(t), m, value_t)$;
        update($t$);
    else if $type_t = V$ then
        update($t$);
        for $v \in keys(subtrees_t)$ do
            OPTBUILD($subtrees_t[v], m \wedge (attr_t : v)$);
    else if $type_t = T$ then
        BUILD($t_-, m$);
        $m_t = m \wedge (attr_t : value_t)$;
        if $needsBarrier(t)$ then
            $emitRule(priority(t), m_t,$ ToController$)$;
        update($t$);
        OPTBUILD($t_+, m_t$);
```

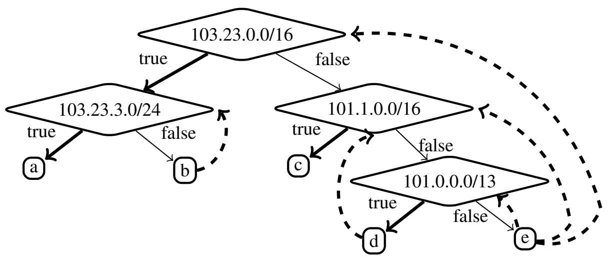


Figure 5: Trace tree from an IP prefix matching example.

An example trace tree of this program is shown in Figure 5, which also shows the G_o edges for this TT. By avoiding barrier rules and by using the dependency graph to limit priority increments, OPTBUILDFT generates the following, optimal prioritization:

```
2, ipDst:103.23.3.0/24 --> a
2, ipDst:101.1.0.0/16  --> c
1, ipDst:101.23.0.0/16 --> b
1, ipDst:101.0.0.0/13  --> d
0, *                   --> e
```

4.4 Efficient Insertion and Invalidation

We now adapt the algorithms of the preceding sections to become incremental algorithms, which typically update rules by examining only a small portion of a trace tree, rather than compiling the entire trace tree "from scratch". Maple allows efficient updates because of information maintained in G_o, which we call node annotations of the tree.

First consider augmenting a trace tree with a new trace. We modify AUGMENTTT to accumulate the identities of $\mathbf{T}$ nodes when the trace follows the negative branch. After attaching the new trace, we build rules starting with the priority stored at the attachment point. Since building rules for the new trace may increase the priorities of some of the $\mathbf{T}$ nodes accumulated above, we modify augmentation to backtrack towards the root and rebuild any positive sub-branches of those $\mathbf{T}$ nodes along this path whose priorities have increased. Note that the method avoids recompiling branches at $\mathbf{V}$ nodes other than the branch being augmented. It also avoids recompiling negative branches of positive $\mathbf{T}$ ancestors of the augmentation point.

Algorithm 6: ROUTEAGGREGATION(ft)

```
1 Safe elimination of rules in $ft$ that have action ToController;
2 for each destination $d$ mentioned in $ft$ do
3     $act =$ action of lowest priority rule in $ft$ overlapping destination $d$ ;
4     $p =$ priority of highest priority rule that overlaps packets to $d$ and agrees with $act$ and such that all lower priority rules overlapping with packets to $d$ agree with $act$ ;
5     Delete rules at priority $\leq p$ matching on destination $d$ from $ft$;
6     $emitRule(p, matchForDest(d), act)$;
```

Invalidation is also simple, because the node annotations provide sufficient information to reconstruct the flow rules generated through a series of updates, even if a full compilation would assign priorities differently. Hence the invalidation of a part of a tree simply involves obtaining the previously-generated rules for that part, leaving all other parts of the tree intact.

4.5 Optimization for Distributed Flow Tables

Maple further optimizes flow table usage through network-wide optimizations, by considering network properties. Below we specify two such optimizations that Maple conducts automatically.

Elimination of ToController **at network core**: To motivate the idea, consider converting the global trace tree in Figure 2 to switch trace trees. Consider the trace tree for a switch s4, which is not on the path from host 6 to 4. If a packet from 6 to 4 does arrive at switch s4, a general, safe action should be ToController, to handle the exception (*e.g.*, due to host mobility). Now Maple uses network topology to know that switch s4 is a core switch (*i.e.*, it does not have any connection to end hosts). Then Maple is assured that the exception will never happen, and hence there is no need to generate the ToController rule for switch s4. This is an example of the general case that core switches do not see "decision misses", which are seen only by edge switches. Hence, Maple does not install ToController rules on core switches and still achieves correctness. Maple can conduct further analysis on network topologies and the trace trees of neighboring switches to remove unnecessary rules from a switch.

Route aggregation: The preceding step prepares Maple to conduct more effectively a second optimization which we call route aggregation. Specifically, a common case is that routing is only destination based (*e.g.*, Ethernet, IP or IP prefix). Hence, when the paths from two sources to the same destination merge, the remaining steps are the same. Maple identifies safe cases where multiple rules to the same destination can be replaced by a single, broader rule that matches on destination only. Algorithm 6 shows the details of the algorithm. The elimination of rules with ToController as action improves the effectiveness of the optimization, because otherwise, the optimization will often fail to find agreement among overlapping rules.

THEOREM 3. ROUTEAGGREGATION *does not alter the forwarding behavior of the network, provided rules at ingress ports include all generated rules with action* ToController.

5. MAPLE MULTICORE SCHEDULER

Even with the preceding optimizations, the Maple controller must be powerful and scalable enough to service the "cache misses" from a potentially large number of switches. Scalability and parallelism within the controller are thus critical to achieving scalability of the SDN as a whole. This section outlines key details of our implementation, in particular, on the scheduler, to achieve graceful scalability. A technical report offers more details [22].

Programming language and runtime: While Maple's architecture is independent of any particular language or runtime infrastructure, its exact implementation will be language-dependent. Our current system allows a programmer to define an algorithmic policy f in either Haskell or Java. In this paper, we focus on implementing f in Haskell, using the Glasgow Haskell Compiler (GHC). We explore the scheduling of f on processor cores, efficiently managing message buffers, parsing and serializing control messages to reduce memory traffic, synchronization between cores, and synchronization in the language runtime system.

Affinity-based, switch-level scheduling of f on cores: Consider application of f at the controller on a sequence of packets. One can classify these packets according to the switches that originated the requests. Maple's scheduler uses affinity-based, switch-level parallel scheduling. Processing by f of packets from the same switch is handled by a single, lightweight user-level thread from a set of user-level threads atop of a smaller set of CPU cores [11, 12]. The scheduler maintains affinity between lightweight threads and hardware cores, shifting threads between cores only in response to persistent load imbalances. This affinity reduces the latency introduced by transferring messages between cores, and enables threads serving "busy" switches to retain cached state relevant to that switch.

This design provides abundant fine-grained parallelism, based on two assumptions. First, a typical network includes many more switches than the controller has cores. Second, each individual switch generates request traffic at a rate that can easily be processed by a single processor core. We expect these assumptions to hold in realistic SDN networks [14].

Achieving I/O scalability: Bottlenecks can arise due to synchronization in the handling of I/O operations. To achieve I/O scalability, Maple further leverages its thread-per-switch design by ensuring that each OS-level thread invokes I/O system calls (`read`, `write`, and `epoll` in this case) on sockets for switches currently assigned to that particular core. Since the scheduler assigns each switch to one core at a time, this affinity avoids I/O-related contention on those sockets both at application level and within the OS kernel code servicing system calls on those sockets.

Further, Maple internally processes messages in batches to handle the issue that an OpenFlow switch typically sends on a TCP stream many short, variable-length messages preceded by a length field, potentially requiring two expensive `read` system calls per message. By reading batches of messages at once, and sending batches of corresponding replies, Maple reduces system call costs.

Large batched `read` operations often leave the head of a switch's message stream in between message boundaries. This implies that using multiple user-level threads to process a switch's message stream would cause frequent thread stalls while one thread waits for the parsing state of a previous parsing thread to become available. Large non-atomic `write` operations cause similar problems on the output path. However, since Maple dedicates a thread to each switch, its switch-level parallelism avoids unnecessary synchronization and facilitates efficient request batching.

6. EVALUATIONS

In this section, we demonstrate that (1) Maple generates high quality rules, (2) Maple can achieve high throughputs on augmentation and invalidation, and (3) Maple can effectively scale controller computation over large multicore processors.

6.1 Quality of Maple Generated Flow Rules

We first evaluate if Maple generates compact switch flow rules.

Algorithmic policies: We use two types of policies. First, we use a simple data center routing policy named `mt-route`. Specifically, the network is divided into subnets, with each subnet assigned a

/24 IPv4 prefix. The subnets are partitioned among multiple tenants, and each tenant is assigned its own weights to network links to build a virtual topology when computing shortest paths. Upon receiving a packet, the `mt-route` policy reads the /24 prefixes of both the source and the destination IPv4 addresses of the packet, looks up the tenants of the source and the destination using the IP prefixes, and then computes intra-tenant routing (same tenant) or inter-tenant routing (*e.g.*, deny or through middleboxes).

Second, we derive policies from filter sets generated by Classbench [20]. Specifically, we use parameter files provided with Classbench to generate filter sets implementing Access Control Lists (ACL), Firewalls (FW), and IP Chains (IPC). For each parameter file, we generate two filter sets with roughly 1000 and 2000 rules, respectively. The first column of Table 1 names the generated filter sets, and the second indicates the number of filters in each Classbench-generated filter set (except for the `mt-route` policy, which does not use a filter set). For example, `acl1a` and `acl1b` are two filter sets generated from a parameter file implementing ACL, with 973 and 1883 filters respectively. We program an `f` that acts as a *filter set interpreter*, which does the following for a given input filter set: upon receiving a packet, the policy tests the packet against each filter, in sequence, until it finds the first matching filter, and then returns an action based on the matched rule. Since TCP port ranges are not directly supported by Openflow, our interpreter checks most TCP port ranges by reading the port value and then performing the test using program logic. However, if the range consists of all ports the interpreter omits the check, and if it consists of a single port the interpreter performs an equality assertion. Furthermore, the interpreter takes advantage of a Maple extension which allows a user-defined `f` to perform a single assertion on multiple conditions. The interpreter makes one or more assertions per filter, and therefore makes heavy use of $\mathbf{T}$ nodes, unlike `mt-route`.

Packet-in: For each Classbench filter set, we use the trace file (*i.e.*, a sequence of packets) generated by Classbench to exercise it. Since not all filters of a filter set are triggered by its given Classbench trace, we use the third column of Table 1 to show the number of distinct filters triggered for each filter set. For the `mt-route` policy, we generate traffic according to [2], which provides a characterization of network traffic in data centers.

In our evaluations, each packet generates a packet-in message at a variant of Cbench [5] — an Openflow switch emulator used to benchmark Openflow controllers — which we modified to generate packets from trace files. For experiments requiring accurate measurements of switch behaviors such as flow table misses, we further modified Cbench to maintain a flow table and process packets according to the flow table. This additional code was taken from the Openflow reference switch implementation.

Results: Table 1 shows the results. We make the following observations. First, Maple generates compact switch flow tables. For example, the policy `acl1a` has 973 filters, and Maple generates a total of only 1006 Openflow rules (see column 4) to handle packets generated by Classbench to test `acl1a`. The number of flow rules generated by Maple is typically higher than the number of filters in a filter set, due to the need to turn port ranges into exact matches and to add barriers to handle packets from ports that have not been exactly matched yet. Define Maple compactness for each filter set as the ratio of the number of rules generated by Maple over the number of rules in the filter set. One can see (column 5) that the largest compactness ratio is for `acl3b`, which is still only 1.31. One can also evaluate the compactness by using the triggered filters in a filter set. The largest is still for `acl3b`, but at 2.09.

Second, we observe that Maple is effective at implementing complex filter sets with a small number of flow table priorities (column 6). For example, it uses only 9 priorities for `acl1a`, which has 973 filters. The `mt-route` policy uses only one priority level.

Alg. policy	#Filts	#Trg	#Rules	Cmpkt	#Pr	Mods/Rule
mt-route			73563		1	1.00
acl1a	973	604	1006	1.03	9	2.25
acl2a	949	595	926	0.98	85	10.47
acl3a	989	622	1119	1.13	33	2.87
fw1a	856	539	821	0.96	79	17.65
fw2a	812	516	731	0.90	56	10.66
ipc1a	977	597	1052	1.08	81	4.20
ipc2a	689	442	466	0.68	26	6.73
acl1b	1883	1187	1874	1.00	18	5.35
acl2b	1834	1154	1816	0.99	119	5.02
acl3b	1966	1234	2575	1.31	119	6.13
fw1b	1700	1099	1775	1.04	113	18.32
fw2b	1747	1126	1762	1.01	60	7.69
ipc1b	1935	1227	2097	1.08	112	9.49
ipc2b	1663	1044	1169	0.70	31	10.02

Table 1: Numbers of flow rules, priorities and modifications generated by Maple for evaluated policies.

Third, operating in an online mode, Maple does need to issue more flow table modification commands (column 7) than the final number of rules. For example, for `acl1a`, on average, 2.25 switch modification commands are issued for each final flow rule.

6.2 Effects of Optimizing Flow Rules

Maple generates wildcard flow rules when possible to reduce flow table "cache" misses. To demonstrate the benefits, we use the `mt-route` policy and compare the performance of Maple with that of a simple controller that uses only exact matches.

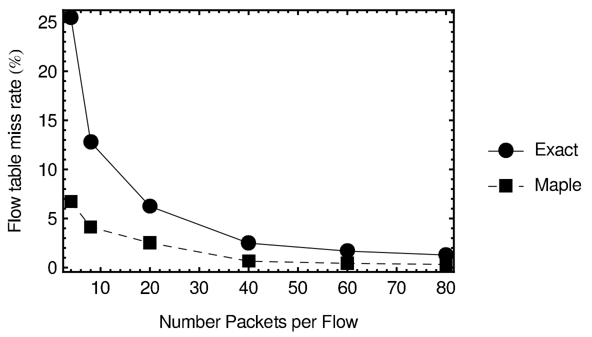
Flow table miss rates: We measure the switch flow table miss rate, defined as the fraction of packets that are diverted from a switch to the controller, at a single switch. We generate network traffic for a number of sessions between 10 hosts and 20 servers, each in distinct tenants, with an average of 4 TCP sessions per pair of hosts. Figure 6(a) shows the flow table miss rates of Maple compared with those of the exact-match controller, as a function of the number of TCP packets per flow, denoted F. We vary F from 4 to 80 packets per flow, as the size of most data center flows fall in this range [2].

As expected, the exact match controller incurs a miss rate of approximately $1/F$, for example incurring 25.5% and 12.8% miss rates for $F = 4$ and $F = 8$, respectively. In contrast, Maple incurs a miss rate 3 to 4 times lower, for example 6.7% and 4.1% at $F = 4$ and $F = 8$. Maple achieves this improvement by generating rules for this policy that match only on source and destination addresses, which therefore decreases the expected miss rate by a factor of 4, the average number of flows per host pair.

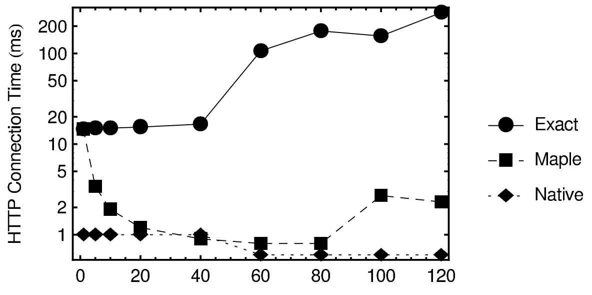
Real HP switch load: We further measure the effects using 3 real HP 5406 Openflow switches (s1, s2, and s3). We build a simple topology in which s1 connects to s2 and s2 connects to s3. A client running httperf [17] at subnet 1 connected at s1 makes HTTP requests to an HTTP server at subnet 3 connected at s3.

We use three controllers. The first two are the same as above: exact match and `mt-route`, which modified to match only on IP and transport fields to accommodate the switches' restrictions on which flows can be placed in hardware tables. We interpret an exact match as being exact on IP and transport fields. We introduce the third controller, which is the native L2 mode (*i.e.*, no Openflow) at any of the switches.

Figure 6(b) shows the mean end-to-end HTTP connection time, which is measured by httperf as the time between a TCP connection is initiated to the time that the connection is closed, and hence includes the time to set up flow tables at all 3 switches. The x-axis of the figure is the request rate and number of requests from the client to the HTTP server. For example, 100 means that the client issues one HTTP connection per 10 ms (=1/100 sec) for a total of 100



(a) Flow table miss rate



(b) HTTP connection time using real HP switches

Figure 6: Effects of optimizing flow rules.

connections. Note that the y-axis is shown as log scale. We make the following observations. First, the HTTP connection setup time of the exact-match controller and that of Maple are the same when the connection rate is 1. Then, as the connection rate increases, since Maple incurs table misses only on the first connection, its HTTP connection time reduces to around 1 ms to slightly above 2 ms when the connection rate is between 10 to 120. In contrast, the exact-match controller incurs table misses on every connection and hence its HTTP connection time increases up to 282 ms at connection rate 120. This result reflects limitations in the switches, since the load on the controller CPU remains below 2% throughout the test, and we ensured that the switches' Openflow rate limiters are configured to avoid affecting the switches' performance. Second, we observe that when the connection rate increases from 80 to 100, the switch CPUs becomes busy, and the HTTP connection time starts to increase from around 1 ms to 2 ms. Third, Maple has a longer HTTP connection time compared with native L2 switch, which suggests potential benefits of proactive installation.

6.3 Flow Table Management Throughput

After evaluating the quality of Maple generated flow table rules, we now evaluate the throughput of Maple; that is, how fast can Maple maintain its trace tree and compile flow tables?

Types of operations: We subject Maple to 3 types of operations: (1) *augments*, where Maple evaluates a policy, augments the trace tree, generates flow table updates, and installs the updates at switches; (2) *lookups*, where Maple handles a packet by looking up the cached answer in the trace tree; and (3) *invalidations*, where Maple invalidates part of the trace tree and deletes rules generated for those subtrees. In particular, we evaluate *host invalidations*, which are caused by host mobility and remove all leaf nodes related to a moved host, and *port invalidations*, which support topology updates and remove any leaf node whose action uses a given port.

Workload: We use the same policies and packet traces as in Section 6.1, but we process each packet trace multiple times. In particular, during the first pass, as nearly every packet will cause an augment operation, we measure the throughput of Maple to record and install new rules. During subsequent passes, we measure lookup throughput, as the trace tree has cached results for every packet in the trace. Finally, we perform invalidations for either all hosts used in the trace or for all ports used in the trace

Filter set	Augments/s	Lookups/s	H Invals/s	P Invals/s
mt-route	58719.65	555937	156284	13215
acl1a	1180.74	58631	2491	3378
acl2a	508.40	15931	1151	1745
acl3a	605.34	19348	1569	2332
fw1a	202.17	73433	3828	5487
fw2a	205.52	85013	4629	7250
ipc1a	362.98	18053	1194	1770
ipc2a	621.23	50544	3089	5066
acl1b	666.67	40133	1358	2200
acl2b	245.92	9809	601	924
acl3b	223.75	9749	626	965
fw1b	68.52	32917	1468	2073
fw2b	51.40	25994	1292	1933
ipc1b	142.11	10143	518	846
ipc2b	185.07	17622	928	1934

Table 2: Maple augments/invalids rates. H Invals and P Invals denote host and port invalidations respectively.

Server: We run Maple on a Dell PowerEdge R210 II server, with 16GB DDR3 memory and Intel Xeon E31270 CPUs (with hyper-threading) running at 3.40GHz. Each CPU has 256KB L2 cache and 8MB shared L3 cache. We run CBench on a separate server and both servers are connected by a 10Gbps Ethernet network.

Results: Table 2 shows throughput for each operation type and policy, using a single 3.40 GHz core, with a single thread. For the `mt-route` policy, which uses only **V** nodes, Maple can perform all operations at high-speed, including both augments and invalidates. The augmentation throughput of Classbench-based policies varies. The `fw2b` policy takes the longest time (20 ms) for Maple to handle a miss. For most policies, invalidation can be handled faster than augmentation, reflecting the fact that invalidations do not require adjusting priority levels, and thus can be done faster.

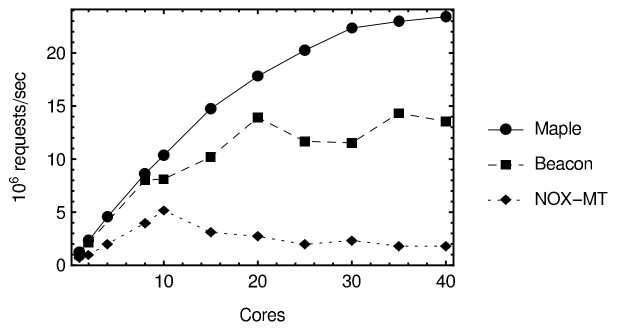
6.4 Run-time Scheduler

We next evaluate the performance of our multicore scheduler. In particular, if the programmer-provided function `f` has no locality, then all requests will be forwarded to the controller for centralized processing. We use learning switch with exact match to evaluate our scheduler, since this controller is available in other frameworks. We measure both throughput (*i.e.*, the number of requests that our controller can process each second) and latency. *The optimizer component of Maple is not executed in evaluations in this section* to compare with Beacon and NOX-MT [21], two well-known Openflow control frameworks that aim to provide high performance.

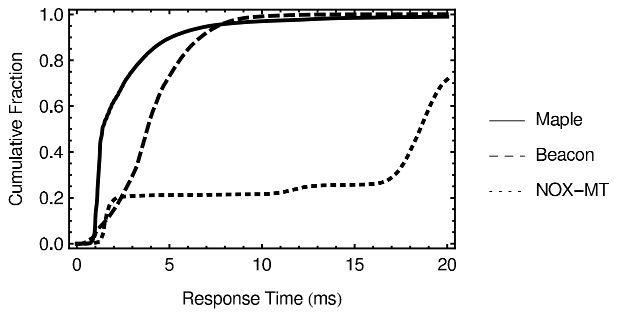
Server: We run our Openflow controllers on an 80 core SuperMicro server, with 8 Intel Xeon E7-8850 2.00GHz processors, each having 10 cores with a 24MB smart cache and 32MB L3 cache. We use four 10 Gbps Intel NICs. Our server software includes Linux kernel version 3.7.1 and Intel ixgbe driver (version 3.9.17).

Workload: We simulate switches with a version of Cbench modified to run on several servers, in order to generate sufficient workload. We use 8 Cbench workload servers connected over 10Gbps links to a single L2 switch, which connects to four 10Gbps interfaces of our control server. We limit the packet-in messages generated by CBench, so that the number of requests outstanding from a single CBench instance does not exceed a configurable limit. This allows us to control the response time while evaluating throughput.

Results: Figure 7(a) shows the throughput as a function of the number of cores used for all three systems. We observe that Maple serves over 20 million requests per second using 40 cores and scales substantially better than Beacon or NOX-MT. In particular Beacon



(a) Throughput comparison



(b) Latency comparison

Figure 7: Throughput and latency of SDN controllers.

scales to less than 15 millions/second, and NOX-MT is only around 2 millions/second. Figure 7(b) shows the corresponding latency CDF for all three systems. The median latency of Maple is 1 ms, Beacon is almost 4 ms, and NOX-MT reaches as high as 17 ms. The 95-percentile latency of Maple is still under 10 ms.

7. RELATED WORK

SDNs have motivated much recent work, which we classify into basic SDN controllers, programming abstractions, offloading work to switches, and controller scalability.

Basic SDN controllers: NOX [8] offers C++ and Python APIs for raw event handling and switch control, while Beacon [1] offers a similar API for Java. These APIs require the programmer to manage low-level Openflow state explicitly, such as switch-level rule patterns, priorities, and timeouts. Maple derives this low-level state from a high-level algorithmic policy expression.

SDN programming abstractions and languages: Maestro [3] raises the abstraction level of SDN programming with modular network state management using programmer-defined views. SNAC [19] and FML [9] offer high-level pattern languages for specifying security policies. Onix [10] introduces the NIB abstraction so that applications modify flow tables through reading and writing to the key-value pairs stored in the NIB. Casado *et al.* [4] proposes network virtualization abstraction. Frenetic [7], Pyretic [16] and Nettle [23] provide new languages for SDN programming. Frenetic's NetCore language supports specialized forms of composition, such as between statistics-gathering and control rules. In contrast, Maple is agnostic to the language for expressing policies, and benefits from whatever features (*e.g.*, composition) the language offers.

Offloading work to switches: Devoflow [6] increases scalability by refactoring the Openflow API, reducing the coupling between centralized control and centralized visibility. Frenetic leverages its NetCore language to compile rules for switch flow tables, alleviating the complex challenge of managing flow tables [7, 15]. Maple similarly compiles to switch flow tables, but its tracing approach supports generic algorithms expressed in arbitrary languages. Also, Maple designs an optimizer that minimizes the number of priority levels, important for reducing update overhead.

SDN controller scaling: The NOX-MT branch of NOX uses Boost for IO and threading. Beacon [1] uses Java threads to scale to a modest number of cores. Onix [10] partitions network state across multiple distributed controllers, alleviating scalability and fault-tolerance concerns by compromising the attractive simplicity of the centralized model. Maestro [3] offers multiple techniques to scale across multiple cores. Maple uses techniques such as affinity-based, switch-level scheduling to achieve substantial scalability.

8. CONCLUSIONS

This paper explores a powerful, programmer-friendly SDN programming model in which a programmer uses standard algorithmic programming to design arbitrary algorithms for SDN control. We present novel techniques to address the scalability challenges of algorithmic SDN programming transparently, and show that the result is highly scalable on a variety of benchmarks using both simulated and real network workloads.

Acknowledgements: We thank Michael F. Nowlan, Lewen Yu, Ramki Gummadi, Erran Li, Haibin Song, Chen Tian, SIGCOMM reviewers, and Ion Stoica (shepherd) for suggestions. Andreas Voellmy was primarily supported by a gift from Futurewei, and partially supported by NSF grant CNS-1017206. Paul Hudak was supported in part by a gift from Microsoft Research.

9. REFERENCES

[1] https://openflow.stanford.edu/display/Beacon/Home.

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

[3] Z. Cai, A. L. Cox, and T. S. Eugene Ng. Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane. Technical report, Rice, 2011.

[4] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *Proc. of PRESTO*, 2010.

[5] Cbench. Cbench, 2012. [Online; accessed 10-April-2012].

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. of SIGCOMM*, 2011.

[7] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a Net. Programming Language. In *Proc. of ICFP*, 2011.

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

[9] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *Proc. of WREN*, 2009.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a Distributed Control Platform for Large-scale Production Networks. In *Proc. of OSDI*, 2010.

[11] S. Marlow, S. Peyton Jones, and S. Singh. Runtime Support for Multicore Haskell. In *Proc. of ICFP*, 2009.

[12] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos. First-class User-level Threads. *ACM SIGOPS Operating Systems Review*, 1991.

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

[14] J. C. Mogul et al. DevoFlow: Cost-effective Flow Management for High Performance Enterprise Networks. In *Proc. of Hotnets*, 2010.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *Proc. of POPL*, 2012.

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

[17] D. Mosberger and T. Jin. httperf: a Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.*, 1998.

[18] D. Shah and P. Gupta. Fast Updating Algo. for TCAMs. *IEEE Micro*, 2001.

[19] Simple network access control (SNAC). http://www.openflow.org/wp/snac/.

[20] D. E. Taylor and J. S. Turner. Classbench: a Packet Classification Benchmark. *IEEE/ACM Trans. Networking*, 2007.

[21] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On Controller Performance in Software-Defined Networks. In *Hot-ICE*, 2012.

[22] A. Voellmy, B. Ford, P. Hudak, and Y. R. Yang. Scaling Software-defined Network Controllers on Multicore Servers. YaleCS TR1468, 2012.

[23] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *Proc. of PADL*, 2011.