Flow-level State Transition as a New Switch Primitive for SDN

Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, Ramesh Govindan
University of Southern California

ABSTRACT

In software-defined networking, the controller installs *flow-based rules* at switches either proactively or reactively. The reactive approach allows controller applications to make dynamic decisions about incoming traffic, but performs worse than the proactive one due to the controller involvement. To support dynamic applications with better performance, we propose FAST (Flow-level State Transitions) as a new switch primitive for software-defined networks. With FAST, the controller simply preinstalls a state machine and switches can automatically record flow state transitions by matching incoming packets to installed filters. FAST can support a variety of dynamic applications, and can be readily implemented with today's commodity switch components and software switches.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]; C.2.1 [**Network Architecture and Design**]; C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software-defined Network; State Machine

1. INTRODUCTION

Software-defined networking has changed networking by separating the control plane from the data plane. While there have been many innovations on controller applications for different network management needs, most of these applications still rely on flow-based rules in the data plane. These flow-based rules often match on multiple packet header fields (e.g., source/destination IP addresses), take predefined actions on matching packets (e.g., dropping the packet, forwarding it to an outgoing port), or maintain counters (e.g., the number of packets or bytes).

The controller saves flow-based rules to switches in two modes: proactive and reactive. In the *proactive* approach, the controller populates rules in switches ahead of time for all the flows coming to the switch. However, the proactive approach requires a priori knowledge of events at switches, and how to handle these events.

HotSDN'14, August 22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2620728.2620729 .

The *reactive* approach supports more dynamic applications, but has poor performance. In the reactive approach, switches often send events (e.g. the first packet of each flow) to the controller, and the controller installs flow-based rules based on these events. However, this introduces significant overhead (CPU, memory, etc.) at the switch, high performance overhead (i.e., delay and throughput), and scalability problems due to the limited communication channel between the controller and switches [13]. For example, consider a stateful firewall that denies unsolicited inbound traffic if it cannot find a corresponding outbound flow in the Established state. The controller becomes aware of the state of outbound flows by receiving the TCP signals (e.g., SYN, FIN) from the switch and denies the unsolicited inbound flows and installs forwarding rules for others upon receiving their first packet. This means that switches have to send *multiple* packets of the same flow to the controller, and the controller has to reactively change the flow-level rules based on these incoming packets.

To reduce the controller involvement in dynamic applications, many works recognize the limitations of flow-based rules and have proposed specific optimizations in the data plane. DevoFlow [13] reduces the controller overhead by introducing rule cloning and measurement triggers. OpenFlow 1.3 supports rate limiting by allowing switches to track flow rates and tag/drop excess traffic without the controller involvement. Open vSwitch [3] adopts the learn action for software switches that can install new rules when traffic matches an old rule.

Instead of proposing yet another specific optimization, we aim at identifying a new generic data plane abstraction to replace flow-based rules. We observe that many networking tasks can be expressed as local state machines over a flow or an aggregate of flows. For example, to implement the stateful firewall, the controller may install a state machine on the switch to keep track of TCP states. The associated action on each state can allow/deny inbound traffic based on the TCP state.

We propose FAST (Flow-level State Transitions) as a new switch abstraction. FAST allows the controller to proactively program state transitions, and allows switches to run *dynamic* actions based on local information. FAST supports a wide range of dynamic applications and can be easily implemented with today's commodity switch components.

FAST includes three parts: (1) an abstraction that allows operators to program their state machines for a variety of applications; (2) a FAST controller that translates state machines to the data plane API and manages the interaction of local state machines with network-wide policies; (3) a FAST data plane that includes a pipeline of tables to support state machines with commodity switch components.

2. MOTIVATING EXAMPLES

Many networking tasks can be expressed as switch-local state machines defined over a flow or an aggregate of flows. The proactive approach cannot express such tasks as their rules change over time; such tasks can be realized using a reactive approach (at the expense of performance) or using FAST (Table 1).

Intrinsic state machines: There are many state machine descriptions whose state definitions are based upon protocol states (e.g., TCP states); we call these intrinsic state machines. For example, stateful firewalls allow all outgoing traffic from a protected machine, but only allow incoming packets if they are part of the response to an outgoing traffic. To only permit those incoming traffic for active TCP connections, a switch acting as a stateful firewall would need to keep track of the TCP states: from the start of a new connection (i.e., receiving the SYN packet), getting SYN-ACK, to the Established state (i.e., receiving the ACK). If the connection is Established, the switch can then forward all the incoming packets. Similarly, the switch should also track the connection state transiting from Established to Closed, in order to remove future incoming packets. In the reactive approach, we must send the TCP signals to the controller to let it track TCP states and install/uninstall corresponding rules. FAST tracks TCP states at the switch and applies the appropriate routing actions to the legitimate connections without involving the controller. Other examples include FTP monitoring, connection affinity in load balancing, QoS in multimedia streaming and link failure recovery (Table 1).

Extrinsic state machines: More complex state machines define states based on additional dynamically-generated information (stored in, for example, counters or bitmaps): we call these *extrinsic* state machines. Examples of tasks that require extrinsic state machines include heavy hitter detection, super-spreader detection, sampling based on flow size and application-aware load balancing (Table 1). For the heavy hitter detection in the reactive approach, the controller must install counting rules at switches and periodically fetch their statistics to detect heavy hitters. The unnecessary communication with the controller wastes bandwidth and switch CPU. Instead, FAST tracks the size of flows and only reports them to the controller when their size reaches a threshold (See Section 4.2 for details of comparison against a threshold.)

As another example, consider a detector for an optimistic ACK attack. To initiate an optimistic ACK attack [29], an attacker at the TCP receiver end often sends sequence numbers for the packets it has not received yet. In this way, the attacker can fool the sender to send data faster. To detect such attack, we can maintain a bitmap to keep track of the sequence numbers of packets that have been sent through the switch. Each time the switch receives an ACK at the reverse direction, we can compare it with the bitmap to identify those ACKs that do not match previous sequence numbers. Without FAST, the switch would need to forward all packets to the controller, and the controller is responsible of maintaining the bitmap and installing a deny rule for detected attacks. Or, we have to rely on an intrusion detection middlebox to maintain such hash table. More complex scenarios that may use multiple variables in each state are also possible for example to detect the application based on the features of the first N packets of a flow [26].

Note that although some of the above examples can be implemented with middleboxes, middleboxes are expensive, lack a general programming interface and usually only support a specific purpose. Instead, we aim at a general flexible solution that can be implemented in commodity switches.

FAST motivation: To support the above examples with commodity switch components, we propose FAST, a new switch primitive based on state machines. FAST allows switches to automatically decide the actions based on the local states. Therefore, FAST achieves better performance for many tasks, and improves the scalability of the SDN system by offloading some work from the controller to switches. Besides performance gains, FAST abstraction allows applying verification tools [12] on the network policies to make sure they match the operator intent. FAST rethinks the boundary between the control and data planes in SDN with the goal of improving performance and scalability with minimal changes to today's switches.

```
1) Task:=(StateMachine,InstanceMapping)
2) StateMachine:=({State},{Transition},{Action},Filter)
3) State:=(name,{Variable})
4) Variable:=(name,#bits)
5) Transition:=(StartState,Condition,TargetState,F)
6) Condition:=f1(StartState.Variables,Packet)→True|False
7) F:=f2(State.Variables,Packet)→TargetState.Variables
8) Action:=(State,Condition,Instruction,Priority)
9) Filter:=f3(Packet)→True|False
10) InstanceMapping:=f4(Packet)→Index
```

Table 2: FAST abstraction

3. FAST ABSTRACTION

FAST provides a state machine processing abstraction over packets in the network. Table 2 describes how a task is defined in FAST using state machines. The parentheses represent a tuple, curly brackets show a set, and right arrows define the output of a function. Each task involves a state machine definition and a mapping for its instances. We now highlight the important aspects of the task definition.

State: Extrinsic state machines require storing counters that represent many states over the variable values. Depending on how the transitions update the variables and the conditions match on them, they can be used as counters, bitmaps or timestamps. Adding the variable to the definition of states instead of making individual states for each value makes the state machine definition much simpler. Later, the controller can map the state names and their variables to a bit string representing actual states.

Transition: A transition from the current state to a target state occurs only if its guard condition (f_1 in Table 2) is true. When the guard is true, the variables in the target state will be set using a function (f_2) over packet fields and the current state variables.

Action: An action executes an instruction on the packet when its guard condition is true. Instruction is defined the same as OpenFlow 1.3. This design allows state machines to accommodate context-specific actions: for example, in a stateful firewall, both inbound and outbound flows match a state in the state machine, but their output port is different, so the associated action may be different.

In FAST, to simplify the semantics of state machines and their implementation, we do not permit a given packet to be processed more than once by a state machine. That is because adding this facility can add unbounded delay to packet processing. Finally, an instruction for a set of flows can be shared across many states, and a compiler can optimize their resource usage based on the switch capabilities (See Section 4).

Filter: The programmer can also filter the traffic going through a set of state machines. For example, it is not necessary to pass UDP traffic through a state machine that tracks TCP states.

Instance mapping: There can be many instances of a state machine in a switch, each in a different state. Thus, the task definition must specify a mapping of each packet to an instance to find its current state. Separating the state machine definition and its instances vs. defining it over all instances simplifies the task definition. Each

	Example	Description	Reactive overhead	FAST state machine
Intrinsic state machines	Stateful firewall	Filter unsolicited inbound TCP connections without any outbound flow	Delay of sending TCP signals for each flow to the controller and installing rules	TCP state machine with actions that drop uninitiated flows
	FTP monitoring	Only allow inbound FTP data channels set up by FTP control channel	Overhead of sending control traffic to the controller to (un)install rules for data channel	Track the states of control channel & allow data channel traffic
	Connection affinity in load balancing	Avoid disrupting ongoing TCP connections during the transition of load between servers	BW overhead of sending old connections to the controller to route to old servers	TCP state machine to distinguish ongoing connections from new ones
	QoS in multimedia streaming	Track states in RTSP or SIP to manage BW/queue reservations	Delay of sending protocol signals to the controller to map connections to queues	Control channel state machine
	Link failure recovery	Use failure-carrying data packets coming backward to use backup routes for forward path [11]	Packet loses while sending failure-carrying packets to controller & installing new routes	State machine over link status
Extrinsic state machines	Heavy hitter detection & rate limiting	Send info of heavy hitters to the controller or drop their excess traffic for rate limiting	BW overhead of periodic query of counters	Keep a counter per flow and compare it against a threshold
	Super-spreader detection	Count open TCP flows from a source to detect super-spreaders (too many connections) [24]	Send all SYN and FIN signals to the controller	Increase a counter on SYNs and decrease it on FINs. Compare the counter to a threshold.
	Sampling based on flow-size	Update the packet sampling rate of a flow based on its estimated size [23]	BW overhead as the controller must periodically fetch counters for each flow and change its sampling rate	Keep a counter for a flow size and select sampling rate based on the counter value
	Accurate flow size distribution	Collect flow sizes to calculate flow-size distribution	Accuracy loss and BW overhead of sampling [22]	Use counters to keep flow sizes and send to the controller upon receiving the FIN signal
	Application-aware load balancing	Track a notion of load for each server based on weights per application request & balance load	Send requests and replies to the controller to track loads	Increase a counter for servers based on the weight of requests and decrease on answers
	Selective packet dropping	Drop differentially-encoded B frames in an MPEG encoded stream if the dependency (preceding I frame) was dropped	BW overhead of sending all packets to the controller	Use a bitmap in states to keep track if the few recent I frames that have been received
	Detect an optimistic ACK attack	Prevent an attacker at TCP receiver to get data faster by sending optimistic ACKs [29]	Send every packet to the controller	Keep a bitmap showing sequence number of packets sent and filter unsolicited ACKs
	Application detection	Detect application type based on features (e.g., size) of the first N packets [26]	BW overhead of sending the first N packets of each connection to the controller	A state machine at the switch looks at features of the first N packets

Table 1: Motivating examples describe the overhead of reactive approach and the state machines in FAST

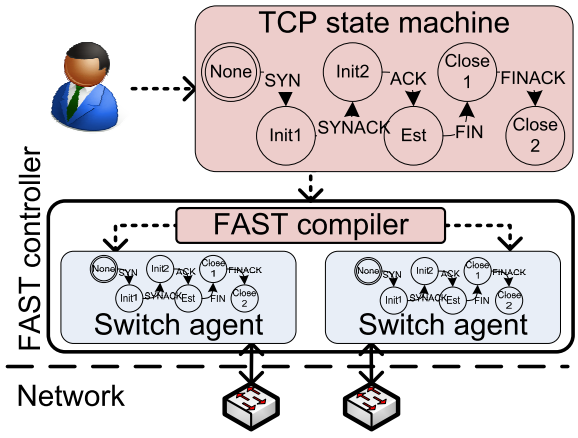

Figure 1: FAST architecture
(Dashed arrows are compile time. Solid arrows are runtime.)

instance will be identified by an index, and a function over packets (f_4) will find it. For example, for tracking the TCP states, we must map two unidirectional flows into an instance. Here, the function can simply be a hash based on the summation of source and destination IP.

4. DESIGN

Given the state machine abstractions defined by operators (Table 2), FAST *proactively* installs them at switches and thus avoids the controller involvement. FAST includes two key designs: the control plane that automatically translates the high-level abstractions into state machines at individual switches, and the data plane that can be readily implemented with today's commodity switch components.

4.1 Control Plane

FAST control plane involves two components (Figure 1): (1) The FAST compiler that compiles state machines into switch agents. (2) The switch agents that manage the local state machines at individual switches. The FAST compiler is an offline component while the switch agents work in runtime.

FAST Compiler: The FAST compiler translates the state machine definitions that an operator specifies to the actual code (switch agents) that can run state machines at individual switches. It uses the information about topology and switch constraints to make switch agents specific to the switch capabilities and configures them to install the state machines only on a subset of switches (e.g., ingress) [33].

Switch agents: Each switch agent preinstalls the state machines at a switch[1] and can communicate with it during the state machine execution. A switch agent has three responsibilities. First, it knows the switch features and how the switch supports the FAST abstraction in data plane. It uses this to convert the state machine to the switch API to perform the state machine functionality. We describe a data plane implementation using a hash table and tables of flow-based rules in Section 4.2.

Secondly, it may perform part of the state machine implementation for the switches with limited capabilities such as features or memory. This means that the switch agent can fall back to the reactive approach and receive packet-ins from switches or fetch statistics periodically. For example, it is easy for software switches to flexibly perform arithmetic computations for the conditions of state transitions while hardware switches may only support wildcard matching. If the switch cannot compute the average flow rate using a counter, the switch agent must periodically fetch the counter to compute the flow rate and apply the rate limiting policy. Moreover, the switch agent can save the state machine partially at the switch to address its limited memory. Because states have different rates of usage (e.g., exception handling states are rare), the switch agent may merge multiple rare states to create a phantom state; whenever, a flow enters such a state, the switch forwards the packet to the controller and the switch agent handles it.

Finally, the switch agent reports local events to the global tasks running at the controller. For example, in Hedera-like traffic engineering [5], the traffic congestion at one link can trigger routing changes at another switch, thus it cannot be specified as a local task. However, the switch can still detect heavy hitters, and the switch agent can configure the switch to send those events to it. Then the switch agent acts like a proxy for the events in the network by passing the events to the global tasks at the controller.

[1] We discuss the reactive installation of state machines in Section 6

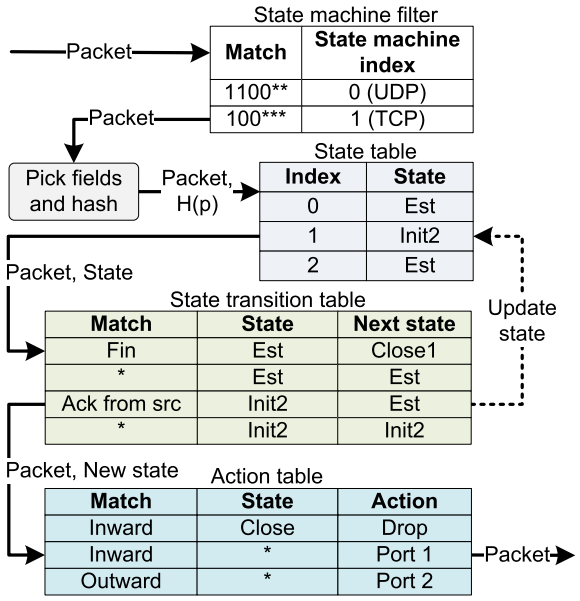

Figure 2: Implementing TCP state machine in FAST data plane

4.2 Data Plane

Figure 2 describes FAST data plane on top of the current flow-based multi-table architecture. While this is not the only architecture to support state machines, especially, in software switches, we believe it is a transition step from the flow-based rules to a state machine based data plane. We use the TCP state machine as a running example to illustrate the components of state machines, which can be implemented with hardware currently available on commodity switches.

The design contains four tables. The state machine filter table is shared among multiple types of state machines, but the other three are specific to each state machine definition. The state table keeps track of state machine instances and is implemented as a hash-table. We decouple actions from state transitions for two reasons. First, it can be more compact: a single entry in action table can cover the actions in multiple states. For example, regardless of whether the state machine is in the Established or Init1 state, one can specify the same action (e.g., forward to port 1). More important, however, is that we *need* this decoupling for our problem domain. While there is one instance of a TCP state machine, the action corresponding to a flow being in a specific state may depend on the flow itself (e.g. output port).

The switch agent installs a state machine on this data plane by adding a rule in the filter table, specifying the hash fields and the initial state for the state table, and installing state transition rules and action rules in the corresponding tables. We now describe the details of each table:

State machine filter: An entry in the state machine filter table corresponds to the *Filter* part of the FAST abstraction (Table 2) which selects the traffic for a type of state machines.[2] This table is identical to conventional OpenFlow tables with actions that point to a state table; it can thus be implemented using TCAMs.

State table: Although all the flows matching a filter rule belong to the same type of state machine, each individual flow corresponds to an instance that may be in a different state. The state table records the current state of each flow and corresponds to the *InstanceMapping* in FAST abstraction. We first pick the packet fields that define the flow (e.g., source IP, destination IP, or even fields in the payload), hash the packet fields, and then track the current flow state in the corresponding entry. The variables inside a state will also be stored in this table. Concretely, the state and the variables map to a number. A few high bits of the number represent the state and others show the current value of the variables.

[2]We can match a flow to multiple state machines by chaining them (Section 6).

Approach	Mean	5^{th}	95^{th}
Proactive	1.85	1.45	3.68
Reactive	84.8	57.84	109.7
FAST	3.02	1.34	5.93

Table 3: Comparing flow completion delay (ms) for 100 experiments

The state table is basically a hash table that can be easily implemented with SRAM and hash modules. Hash-tables are already used in switches for NetFlow counters [1]. An alternative design is to use regular tables and update the state by inserting rules [6]. However, hash tables allow us to avoid inserting a rule into TCAM which has unpredictable high delay [18]. After finding the entry, the state and the variables can be saved in the packet metadata for next tables. Note that the state in the state table may depend on different flows. For example, for the TCP connection, the state depends on the bidirectional flows. Therefore, we choose to hash on source and destination IPs as two parameters independent of their order, so that bidirectional flows are hashed into the same entry. Different state machines may pick flows based on different packet fields. For example, in super-spreader detection, we only pick source IP to define the state.

State transition table: Each entry in this table represents a transition and contains three distinct entities: the matching on current state, the condition on state variables and packet fields, and the next state. When the packet matches the current state and the conditions, the state will be updated in the state table based on the next state, while the packet will go to the action table carrying the new state as its metadata.

To implement the state transition table, we can install state transition rules in TCAM to match on the current state and packet fields. We can also match on the state variables already carried in the packet metadata. For example, to compare a counter against a threshold, we can translate a transition condition to at most n wildcard matching rules, where n is the number of bits specified for the counter (Table 2)

Action table: The action table matches the new state and packet header fields to a specific action (e.g., forward the packet to a port, or drop the flow). More precisely, each entry in the action table has three components: the new state, flow rules defined on packet header fields, and an action. When a flow matches the state and an incoming packet matches the flow rules, the specified action will be performed. Action tables can also be implemented easily with TCAMs.

5. FAST PROTOTYPE AND EVALUATION

Prototype: We have implemented a stateful firewall in FAST prototype with POX and Open vSwitch [3]. At the controller side, a switch agent proactively installs state machines using OpenFlow protocol. We use Nicira extensions that supports the learn action and hashing, and we added the matching on TCP signals. We have instantiated the state machines and changed their states leveraging the Open vSwitch learn action.

Evaluation setting: We run a FAST controller, a FAST switch, a sender, and a receiver in Mininet [16]. We use the TCP state machine as an example. The state machine transits on SYN, SYNACK, ACK, FIN, and FINACK flags and the switch drops any packet

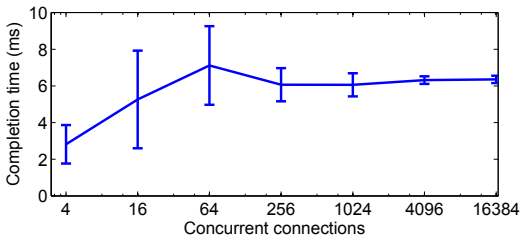

Figure 3: Flow completion for concurrent small flows for 10 experiments

from the receiver if the sender does not expect it as defined in a stateful firewall. We compare FAST with two approaches: (1) the *proactive* approach where the controller just pre-installs two routing rules. This cannot track the states and is used as a baseline for comparison. (2) the *reactive* approach where the controller receives the signals using the following trick: It only installs the routing rules after the connection went to the TCP Established state. It also installs an additional rule along with routing rules to send FIN signals to the controller.

FAST can change states with low delay. To stress FAST under many state changes per flow, we generate a small flow with one data packet and compute the delay of its transitions through the TCP state machine (flow initiation until termination). FAST has much smaller delay comparing to the reactive approach because there is no need to send the signals to the controller (Table 3). However, due to the overhead of calculating the hashes for looking up the current state and updating the state table, its delay is larger than the basic proactive approach.

FAST state-lookup has low overhead. To quantify only the overhead of state lookup, we compare the throughput of FAST vs. proactive approach for a long connection using iperf. The iperf connection remains in the Established state for many packets, so this comparison quantifies only the overhead of hashing and checking the current state. The throughput of FAST is very close to the proactive approach (7.8 Gbps vs 8.2 Gbps, respectively with standard deviation of 0.2Gbps over 30 tries).

FAST is scalable. Finally, we track the flow completion times for many concurrent flows in FAST in order to evaluate FAST data plane scalability. Figure 3 shows that the delay of small flows remains small even for thousands of concurrent flows.

In conclusion, FAST overhead is negligible for many concurrent connections while it is more expressive compared to the proactive approach and has much less delay compared to the reactive approach.

6. CHALLENGES AND FUTURE WORK

Reactive install: The state machine definition can be installed reactively. The switch agent can wait for a switch to find no state machine to handle a packet and send it to the controller. Then it installs a new definition.

Installing consistent state machines: There are two consistency concerns in installing state machines: The consistency concern in installing state machines on multiple switches requires further research to make sure a packet/flow is handled by the same policy scattered among multiple switches (same as flow-based rules [28]). Secondly, locally at a switch, we must define how changing a state machine can reuse the information of its old version. For example, we may create new instances of the new state machine out of the old ones and reuse state variables (e.g., flow size counters).

Verifying state machines: A motivation for choosing state machine abstraction is its prevalence of verification tools in the programming stage [12]. The tools allow comparing the state machine definition against the policy the user expects and making sure it does not have bugs such as overlapping transition conditions or unused transitions. Providing such services using the available tools before compiling the state machines is in our future work. Moreover, to debug the networking tasks online, the controller can dump the current state of the state machine instances to build a complete network view. It can also tune the visibility of events by making switches to send a copy of the packets causing state transitions to the controller.

Composing multiple state machines: With FAST, operators can specify different state machines for different tasks (e.g., one for load balancing and another for stateful firewall). However, a single flow may need to traverse both the load balancer (with the state machine to ensure connection affinity) and the stateful firewall (with the TCP state machine). If matching against multiple rules in the state machine filter table [36] is not possible, a compiler at the controller needs to generate a combined state machine for these tasks. For example, it can either combine their states and transitions or chain state machines by resubmitting the packets to the second state machine from the action table of the first. More generally, the FAST compiler may also explore opportunities of resource sharing across state machines. For example, both the state machine to ensure connection affinity and the TCP state machine for a stateful firewall check the first SYN packet of TCP connections, but perform different actions. The compiler may also verify these state machines to avoid conflicts by using the techniques in recent SDN verification tools (e.g., VeriFlow [20]).

Features at switches: Matching on a richer set of packet fields in switches enables FAST to push more responsibilities to switches in order to support more usecases efficiently: OpenFlow compliant switches can already support limited packet header fields. However, future commodity switches will be able to flexibly parse packet headers [9, 8] and extract the protocol dependent fields such as video frame types or FTP data connection port from the control flow. Indeed, software switches can perform protocol analysis beyond layer 4 and thus enable data-conditional state machines (e.g., by reusing the protocol analyzers in Bro [25] in a flexible API such as Intel DPDK [2]).

Using switch resources efficiently: The switch agents can do several optimizations to fit state machines into the data plane with limited memory. If the matching field of rows in a state transition table with the same next state value are the same (all input links to each state are the same), we can compact a state transition table onto a TCAM table efficiently [10]. We can also use approximate concurrent state machines (ACSM) [7] when reaching a "don't know" state occasionally is acceptable. Besides, the hardware switch itself can offload non-frequent states to CPU-DRAM. For software switches, combining state machines [35] can improve the latency of matching.

7. RELATED WORK

Dynamic actions at switches: DevoFlow [13] proposed rule cloning and trigger based measurement reports at switches to reduce the overhead of reactive approach. OpenFlow 1.3 introduces the meter table that allows using counters for rate-limiting pre-configured by the controller. FAST proposes a more general abstraction that covers the above scenarios. Open vSwitch [3] supports the learn action that allows adding a rule as an action of an OpenFlow rule. In con-

trast, FAST abstraction is more general and FAST data plane design using the hash table also works in hardware switches. Active network [32] and Tiny packets [19] let switches run a small program. FAST limits the program to state machines and only allows the trusted controller to program switches. P4 [8] and RMT [9] introduce flexible header parsing at switches. FAST can leverage these parsers in the data plane.

State machine abstraction: State machines have been identified as an important abstraction for many networking tasks [7, 30]. Even if we do not know the state machine for a network protocol, there are proposals to infer it from traces [34]. Moreover, PyResonance [4] implements a programming language to define state machines at the controller. However, it later translates them to flow-based rules and uses the reactive approach to run the networking tasks. FAST is complementary to above researches by enabling switches to run state machines locally to improve performance. As in FAST, OpenState [6] also proposed a state machine abstraction at the data plane. Although this parallel work is similar in abstraction, its data plane design is different. While OpenState has a state table and an XFSM table (state transition + action table), FAST separates these tables to shrink their size. OpenState also inserts rules in the state table for updating the current state, but FAST uses a hash-table for the state table to avoid the long unpredictable delay of rule insertion. Additionally, in this paper, we looked at more applications (Table 1), discussed the controller design and challenges, and provided preliminary evaluation results.

Middlebox management in SDN: A strand of research tries to redesign middleboxes to let a controller manage them [15, 31]. Another category manages middleboxes with the current implementation [27, 14]. In contrast, FAST pushes the limit of current switches to efficiently implement more network functions than simple forwarding.

Local Controller Logic: Several works at the controller side ([17, 21, 33]) recognized the ability of separating a control policy to local and network-wide logics. We also use this argument to motivate not centralizing the logic at the controller. However, instead of local controllers [17], we believe the state machine abstraction is simple enough to be implemented at fast data planes and general enough to cover many usecases.

8. CONCLUSION

We observe that many networking functions require changing the actions within the same flow based on the current flow state. Thus we propose flow-level state transitions (FAST) as a new primitive for SDN. Compared to the primitive of flow-based rules, FAST supports more flexible networking tasks, improves the performance and scalability of the SDN controller, and can be easily implemented with commodity switch components.

9. REFERENCES

[1] Cisco Catalyst 6500 Supervisor Engine 2T - NetFlow Enhancements White Paper. http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11-652021.html.

[2] Intel Data Plane Development Kit. http://dpdk.org/.

[3] Open vswitch. http://openvswitch.org/.

[4] PyResonance. https://github.com/Resonance-SDN/pyresonance/wiki.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[6] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *CCR*, 44(2):44–51, 2014.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *SIGCOMM*, 2006.

[8] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. Programming Protocol-Independent Packet Processors. *CoRR*, abs/1312.1719, 2013.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[10] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. In *INFOCOM*, 2010.

[11] M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker. Dynamic Route Computation Considered Harmful. *CCR*, 40(2):66–71, 2010.

[12] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, 2002.

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

[14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *NSDI*, 2014.

[15] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-defined Middlebox Networking. In *HotNets*, 2012.

[16] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments Using Container-based Emulation. In *CoNEXT*, 2012.

[17] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.

[18] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In *HotSDN*, 2013.

[19] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières. Tiny Packet Programs for Low-latency Network Control and Monitoring. In *HotNets*, 2013.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.

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

[22] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. *SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[23] A. Kumar and J. Xu. Sketch Guided Sampling - Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *INFOCOM*, 2006.

[24] S. A. Mehdi, J. Khalid, and S. A. Khayam. Revisiting Traffic Anomaly Detection Using Software Defined Networking. In *RAID*, 2011.

[25] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[26] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir. Application-awareness in SDN. In *SIGCOMM*, 2013.

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

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[29] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *CCR*, 29(5):71–78, 1999.

[30] D. V. Schuehler and J. W. Lockwood. A Modular System for FPGA-based TCP Flow Processing in High-speed Networks. In *Field Programmable Logic and Application*. 2004.

[31] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[32] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *CCR*, 37(5):81–94, 2007.

[33] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. *CCR*, 43(4):87–98, 2013.

[34] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo. Inferring Protocol State Machine From Network Traces: a Probabilistic Approach. In *ACNS*, 2011.

[35] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *ANCS*, 2006.

[36] F. Yu, R. H. Katz, and T. Lakshman. Efficient Multi-Match Packet Classification and Lookup with TCAM. *Micro*, 25(1):50–59, 2005.