Tiny Packet Programs for low-latency network control and monitoring

Vimalkumar Jeyakumar[1], Mohammad Alizadeh[2], Changhoon Kim[3], David Mazières[1]
{jvimal,alizade}@stanford.edu, chakim@microsoft.com, www.scs.stanford.edu/∼dm/addr
[1]Stanford University [2]Insieme Networks [3]Windows Azure
Stanford, CA, USA San Jose, CA, USA Redmond, WA, USA

ABSTRACT

Networking researchers and practitioners strive for a greater degree of control and programmability to rapidly innovate in production networks. While this desire enjoys commercial success in the control plane through efforts such as OpenFlow, the dataplane has eluded such programmability. In this paper, we show how end-hosts can coordinate with the network to implement a wide-range of network tasks, by embedding tiny programs into packets that execute directly in the *dataplane*. Our key contribution is a programmatic interface between end-hosts and the switch ASICs that does not sacrifice raw performance. This interface allows network tasks to be refactored into two components: (a) a simple program that executes on the ASIC, and (b) an expressive task distributed across end-hosts. We demonstrate the promise of this approach by implementing three tasks using read/write programs: (i) detecting short-lived congestion events in high speed networks, (ii) a rate-based congestion control algorithm, and (iii) a forwarding plane network debugger.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.3 [**Computer-Communication Networks**]: Network Operations; C.4 [**Performance of Systems**]: Design Studies, Performance Attributes

General Terms

Design, Measurement, Performance

Keywords

Active Networks, SDN, Software-Defined Networks, Network Architecture, Switch ASIC Design

1. INTRODUCTION

Dataplane network tasks such as congestion control, measurement, and fault diagnosis, benefit from low-latency[1] visibility into network state. Many research proposals show how to make better decisions with more visibility; for example, Rate Control Protocol (RCP) [1] is a congestion-control mechanism that uses link utilization and average queue sizes to allocate bandwidth to flows rapidly so they converge quickly to their max-min fair rates. Unfortunately, deploying such proposals requires ASICs that directly implement the required functionality in the dataplane, as the control plane is orders of magnitude slower to provide such visibility. However, designing ASIC features for each task can take many years, and once fabricated, they cannot be changed for years. We are stuck in a world where we have neither the flexibility in the dataplane, nor the required performance to access network state for dataplane tasks.

End-hosts could readily implement many network tasks if they had access to network state. Thus, an ideal goal would be to have flexible access to network state that is fast enough for end-hosts to implement dataplane tasks. To this end, this paper describes a *simple* programmable interface that enables *end-hosts* to query and compute on switch memory (ASIC registers, SRAM, etc.) using packets, directly in the dataplane. Specifically, packets carry a tiny packet program (TPP) in their header, which consists of a few instructions that read, write, or perform arithmetic using data on the ASIC registers and SRAM. We show how this dataplane interface enables end-hosts to rapidly deploy new functionality by refactoring it into: (a) simple TPPs that execute on the ASICs, and (b) expressive tasks at end-hosts.

An astute reader might ask, "Isn't this Active Networking?" Active networking [2, 3] proposed an audacious idea of allowing network routers to execute *arbitrary* programs to carry out tasks that actively control network behaviour such as routing, packet compression, and duplication. Our goal, however, is to explore an alternative that is simple and cost-effective, while still meeting an ASIC's stringent

[1]By low-latency, we refer to timescales on the order of round-trip times; it can be a few microseconds in a datacenter, or 10s of milliseconds in wide-area networks.

Instruction	Meaning
`LOAD, PUSH`	Copy values from switch to packet
`STORE, POP`	Copy values from packet to switch
`CSTORE`	Conditional store for atomic operations
`CEXEC`	Conditionally execute the subsequent instructions

Table 1: The tasks we present in the paper require support only for the above instructions, whose operands will be clear when we discuss examples.

performance requirements.[2] The instructions are simple in that they execute within the time budget for handling small sized packets at line-rate. These instructions free the ASIC of complex tasks, leaving end-hosts to coordinate with the network directly in the dataplane to achieve a desired functionality. We show how even a minimal read and write instruction set enables new capabilities on a network, many of which would otherwise be feasible today only after years of investment on ASIC development.

This interface raises a number of questions and concerns, which is the subject of this paper.

- How general is this interface? In §2, we walk through refactoring three different network tasks on a TPP enabled Linux Router using read/write instructions.
- Can TPPs work at line-rate and what are the overheads? (§3) Restricting TPPs to (say) five instructions *per-packet* requires only 20 bytes of instruction overhead and up to 60 bytes of output space, and execution takes less than a packet's transmission time. Nonetheless, we show how network tasks can use many TPPs to overcome this limitation.
- Where is it applicable, and what are its security implications? (§4) The rise of large-scale and privately owned networks (e.g., datacenters, WANs) makes the TPP approach attractive. In such networks, only trusted entities may use TPPs.

2. EXAMPLE PROGRAMS

In this section we use a sequence of tiny packet programs (TPPs) at end-hosts to implement three network tasks: (i) micro-burst detection, (ii) a rate based congestion control algorithm, and (iii) a network forwarding plane debugger.

What is a TPP? A TPP is any ethernet packet with a uniquely identifiable header that contains instructions, some additional space (packet memory), and encapsulates an optional ethernet payload. The TPP exclusively owns its packet memory, but also has access to shared memory on the switch (its SRAM and internal registers) through a virtual address interface. TPPs are executed on a tiny CPU (TCPU) in the dataplane by the ASIC, but are forwarded just like other packets. TPPs use a very minimal instruction set listed in Table 1. Section 3 talks about the structure of a TPP, the virtual address space, and the TCPU in greater detail.

[2]A 64-port 10GbE switch has to process about a billion 64-byte-packets/second to operate at line-rate.

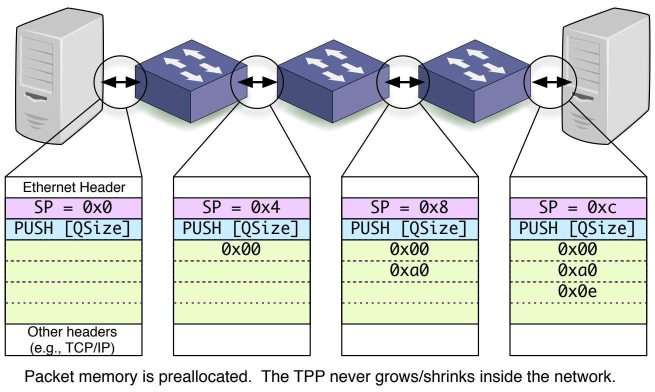


Figure 1: Visualizing the execution of a TPP that queries the network for queue sizes. As the TPP traverses a network of switches, the ASIC executes the program, which modifies the packet to reflect the queue sizes on the link.

For readability, when we write TPPs in an x86-like assembly language, we will refer to specific dataplane statistics using the notation `[Namespace:Statistic]`. For instance, `[Queue:QueueSize]` will be compiled a virtual memory address (say) `0xb000` at compile time. To the ASIC, the address `0xb000` refers to the queue size on the link the packet will be sent out. To simplify discussion, we assume that the address is the same across all network devices, and that they are unaffected by network operation (such as routing).

2.1 Micro-burst Detection

In low-latency networks such as datacenters, queueing delays contribute significantly to overall network latency. Queue occupancy fluctuations due to small-timescale congestion (i.e., "micro-bursts") are hard to detect as queues change at timescales of a few RTTs, which can be as small as a few 100 microseconds. Today's monitoring mechanisms operate only on timescales that are 10s of seconds at best, and are therefore ill-suited for isolating micro-bursts.

TPPs can provide fine-grained per-RTT, or even per-packet visibility into queue evolution inside the network. Today, the ASIC memory manager already keeps track of per-port, per-queue occupancies in its registers. If packet memory is addressed like a stack, then, the instruction `PUSH [Queue:QueueSize]` copies the queue register onto packet memory. As the packet traverses each hop, the packet memory records snapshots of queue size statistics at each hop. The queue sizes are useful in diagnosing micro-bursts, as they are not an average statistic. They are recorded *the instant the packet traversed the switch*. Figure 1 shows how the state of of a sample packet as it traverses a network. In the figure, `SP` is the stack pointer which points to the next offset inside the packet memory where new values may be pushed. Since the maximum number of hops is small within a datacenter (typically 5–7), the end-host preallocates enough packet memory to store queue sizes. Moreover, the end-host knows exactly how to interpret values in the packet to obtain a detailed breakdown of queueing latencies on all network hops.

This example illustrates how a low-latency, programmatic interface to access dataplane state can be used by software at end-hosts to measure dataplane behavior that is hard to observe in the control plane.

2.2 Rate based congestion control

While the previous example shows how TPPs can help debug latency spikes using queue sizes, we now show how such visibility can be used to *control* network congestion. Congestion control is arguably a dataplane task, and the literature has a number of ideas on designing better algorithms. However, TCP and its variants still remain the dominant congestion control algorithms. We show how end-hosts can use TPPs to deploy a new congestion control algorithm that enjoys many benefits of Rate Control Protocol (RCP) [1]. RCP is a congestion control algorithm that rapidly allocates link capacity to help flows finish quickly. An RCP router maintains one fair-share rate per link $R(t)$ (regardless of the number of flows), computed periodically (every T seconds) as follows:

$$R(t+T) = R(t)\left(1 - \frac{T}{d} \times \frac{\alpha\,(y(t)-C) + \beta\,\frac{q(t)}{d}}{C}\right)$$

Here, $y(t)$ is the average ingress link utilization, $q(t)$ is the average queue size, and d is the average round-trip time of flows traversing the link, and α and β are configurable parameters. Each router checks if its estimate of $R(t)$ is smaller than the flow's fair-share (indicated on each packet's header); if so, it replaces the flow's fair share header value with $R(t)$.

We now describe RCP*, an end-host implementation of RCP. The implementation consists of a rate limiter and a rate controller at end-hosts for every flow (since RCP does per-flow congestion control). For ease of exposition, we assume all links have the same capacity, and all flows have the same RTT. Each flow's rate controller periodically (using the flow's packets, or using additional probe packets) queries and modifies network state in three phases.

Phase 1: Collect. Using the following TPP, the rate controller queries the network for the switch ID on each hop, queue sizes, link utilizations, and the link's fair share rate, for all links along the path. The receiver simply echos a fully executed TPP back to the sender.[3]

```
PUSH [Switch:SwitchID]
PUSH [Link:QueueSize]
PUSH [Link:RX-Utilization]
PUSH [Link:RCP-RateRegister]
```

Phase 2: Compute. In the second phase, each sender computes a fair share rate R_{link} for each link: Using the samples collected in phase 1, the rate controller computes the average queue sizes on each link along the path. Then, it computes a per-link rate R_{link} using the RCP control equation.

[3]We assume a control plane program initializes each link's fair share rate to its capacity.

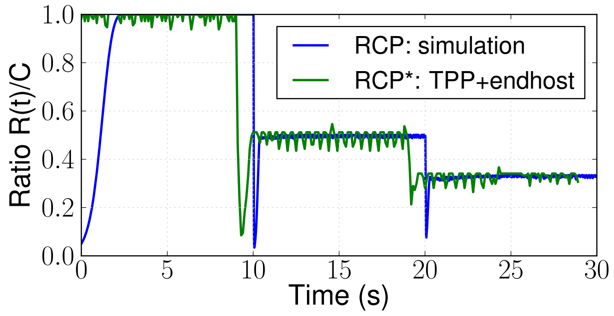


Figure 2: We compare a Linux router based implementation of RCP* and a simulation of the original RCP algorithm. We start one flow each at t=0s, t=10s and t=20s and we find that RCP* helps flows converge quickly to their fair share on the bottleneck link.

Phase 3: Update. In the last phase, since the rate-controller clearly knows the bottleneck link from the values of R_{link} (the minimum), it sends a TPP that only executes on the bottleneck switch link to update its per-link's state. This is achieved using the conditional execute instruction (`CEXEC`), where, `CEXEC reg,mask,value` ensures the TPP executes on a switch only if `(reg & mask) == value`. (Note that the end-host need not know the actual route to reach the bottleneck switch link.)

```
CEXEC [Switch:SwitchID], 0xFFFFFFFF,
         $BottleneckSwitchID
STORE [Link:RCP-RateRegister],
         [PacketMemory:Offset]
```

Each flow executes the above TPP, *independently* writing values into the ASIC's register, and reading it through subsequent TPPs. With multiple concurrent writers to a shared switch memory, one might wonder if there could be race conditions that are hard to detect. While this is a legitimate concern for network tasks such as accounting, we found that congestion control does not require such strong notions of consistency. Nevertheless, we support a conditional store instruction to provide a stronger (linearizable [4]) notion of consistency for memory updates. `CSTORE dst,cond,src` stores `src` into `dst` only if `dst==cond`.

Our RCP* implementation is a userspace program that sends UDP packets to query a TPP enabled Linux Router. Figure 2 plots the evolution of $R(t)/C$ on a 10Mb/s bottleneck link shared by three flows. We compared our implementation with the original RCP algorithm available in ns2 simulation. As we can see, the behavior of RCP and RCP* are qualitatively similar, in that they both show quick convergence (we set $\alpha = 0.5, \beta = 1$ for both).

This example shows the benefits of our proposed refactoring: the ASIC only supports reads and writes, but the end-hosts can use this state to implement a new congestion control mechanism without requiring a specialized ASIC.

2.3 Forwarding plane debugger

There has been recent interest in verifying that network forwarding rules match the intent specified by the administrator [5, 6]. However, forwarding rules change constantly, and a network-wide consistent update is not a trivial task [7]. Forwarding rule verification is further complicated by the fact that there can be a mismatch between the control plane's view of routing state and the actual forwarding state in hardware. Thus, verifying whether *every packet* has been correctly forwarded requires help from the dataplane.

We now show how to implement a network forwarding plane debugger `ndb` [8] for a Software-Defined Network. `ndb` works by interposing on the control channel between the controller and the network, stamping each flow entry with a unique version number, and modifying flow entries to create truncated copies of packet headers tagged with the version number (without affecting a packet's normal forwarding) and additional metadata (e.g., the packet's input/output ports). These truncated packet copies are reassembled by servers to present a unified view of a packet's journey through the network. This trace can then be used for verification.

Using TPPs, end-hosts can get the same level of visibility as `ndb` by having a trusted entity insert the TPP shown below on all its packets. On receiving a TPP that has finished executing on all hops, the end-host gets an accurate view of the network forwarding state that affected the packet's forwarding, without requiring the network to create additional packet copies. This information can then be used by software to verify whether network forwarding conforms to a specified policy.

```
PUSH [Switch:ID]
PUSH [PacketMetadata:MatchedEntryID]
PUSH [PacketMetadata:InputPort]
```

Other possibilities. The above examples illustrate how simple primitives in ASICs can enable end-hosts to coordinate and achieve a certain behavior. TPPs are not just limited to wired networks; they can also be used in wireless networks where access points can annotate end-host packets with channel SNR which changes very quickly. Low-latency access to such rapidly changing state is useful for network diagnosis and fault localization.

3. DESIGNING A PROGRAMMABLE ASIC

In this section, we walk through the design of a TPP-capable ASIC and discuss why it is feasible to process TPPs in the dataplane. The ASIC is one important component in the end-to-end picture of low-latency and programmable network control. Our holistic design is guided by the following design principles:

- **Simplicity in the network.** Switch ASICs have stringent performance requirements to work at line-rate, and therefore, any complex design increases its cost which hinders its adoption. At the very least, implementing a network task needs access to network state, which is exactly what our minimal instruction set provides (besides the ability to synchronize multiple writers).
- **Smartness at the edge.** Any complexity in implementing a network task is pushed to fully programmable end-hosts, which can act on network state to make smart decisions.

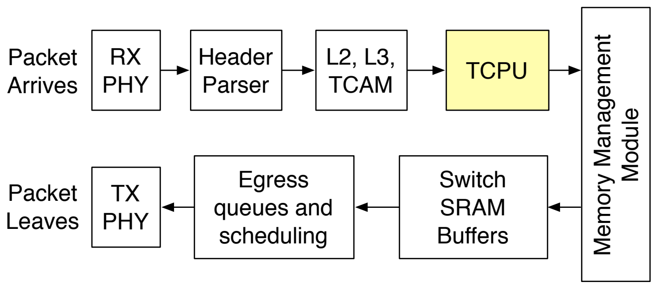


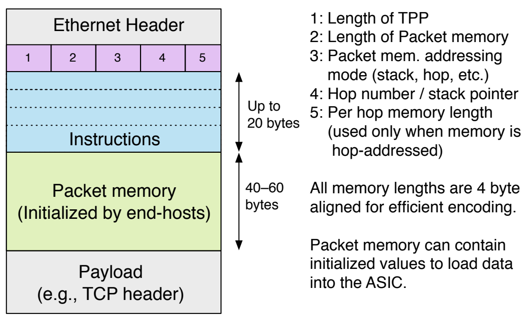
Figure 3: **A simplified block diagram of the dataplane pipeline in a switch ASIC. Packets arrive at the ingress, and pass through multiple modules. The scheduler then forwards packets (that are not dropped) to the output ports computed at earlier stages in the pipeline. The tiny CPU (TCPU) that processes TPPs is placed just before the packet is stored in memory.**

3.1 Background on an ASIC pipeline

Figure 3 shows a simplified block diagram of an ASIC. The packet flows from input to output(s) through many pipelined hardware modules. Here, the PHY module first decodes the packet from the wire, and passes it to a dataplane module, which tags the packet with metadata (such as its ingress port number). Then, the packet passes through a header parser that extracts fields from the packet header and passes it further down the pipeline, which uses the parsed fields to route the packet (using a combination of layer 2 MAC table, layer 3 longest-prefix match table and a flexible TCAM table). Finally, any modifications to the packet are committed and the packet is queued in switch memory. Using metadata (such as the packet's priority), the scheduler decides when it is time for the packet to be transmitted out of the egress port determined earlier in the pipeline. A more thorough pipeline in a commercial ASIC can be found in [9, Page 26] and [10, Page 7].

3.2 Programming Model

Tiny packet programs (TPPs) are executed in the dataplane pipeline. The ASIC parses and sequentially executes these instructions on a tiny CPU (TCPU). The TPP has access to all *switch memory* which includes ASIC registers and SRAM, as well as *packet memory*, which is a small scratch space included within the packet payload (limited by the packet length). End-hosts can use multiple packets if a single packet is insufficient for a network task. Figure 4 shows the structure of a TPP. Unless otherwise noted, a TPP executes at all TCPU-enabled ASICs it traverses.



Ethernet Header
1
2
3
4
5
Instructions
Packet memory (Initialized by end-hosts)
Payload (e.g., TCP header)
Up to 20 bytes
40–60 bytes
1: Length of TPP
2: Length of Packet memory
3: Packet mem. addressing mode (stack, hop, etc.)
4: Hop number / stack pointer
5: Per hop memory length (used only when memory is hop-addressed)
All memory lengths are 4 byte aligned for efficient encoding.
Packet memory can contain initialized values to load data into the ASIC.

Figure 4: **A packet carrying a tiny program.**

Namespace	**Examples**
Per-Switch	Switch ID, counters associated with the global L2 or L3 flow tables, flow table version number [8], etc.
Per-Port	Link utilization, bytes received, bytes dropped, bytes enqueued.
Per-Queue	Bytes enqueued, bytes dropped
Per-Packet	Packet's input/output port, matched flow entry [8], alternate routes for a packet [11].

Table 2: **By having access to the switch's memory address space, the TPP can access statistics listed above. Many statistics are already tracked by today's ASICs, but others, such as flow table version (for `ndb`), will have to be implemented.**

Multiple tasks. We rely on a control-plane agent to partition switch SRAM and isolate concurrently executing network tasks. For instance, if end-hosts implement both RCP and `ndb`, the agent would allocate a non-overlapping set of SRAM addresses to RCP and `ndb`. We defer the discussion of memory allocation to future work.

3.2.1 Unified Memory-Mapped IO

A TPP has access to any switch statistic tracked by the ASIC. The statistics can be broadly namespaced into per-switch (i.e. global), per-port, per-queue and per-packet. Table 2 shows example statistics in each of these namespaces. These statistics reside in different memory banks, but providing a unified address space makes them available to TPPs. For instance, in its registers, the ASIC keeps metadata such as input port, the selected route, etc. for every packet. The memory locations `0xa000 +` {`0x1,0x2`} could refer to the input port and the selected route. These address mappings must be known upfront so that the TPP compiler can convert mnemonics (such as `PacketMetadata:InputPort`) into addresses.

3.2.2 Addressing Packet Memory

Instructions can manage packet memory using x86-like addressing schemes. For example, memory can be managed using a stack pointer and a `PUSH` instruction that appends values to preallocated packet memory. We also found a hop addressing scheme to be useful, which is similar to the the `base:offset` x86-addressing scheme. Here, `base:offset` refers to the word at location `base * size + offset`. Thus, if size is 16 bytes, the instruction `LOAD [Switch:SwitchID], [Packet:hop[1]]` will copy the switch ID into `PacketMemory[1]` on the first hop, `PacketMemory[17]` on the second hop, etc. The numbers `base` (hop number) and `offset` are part of the instruction and the size of the per-hop data structure is kept in the TPP header. To simplify memory management in the ASICs, the end-host preallocate enough space in the TPP to hold per-hop data structures. Recall that end-hosts can use multiple TPPs if one packet is insufficient to load all statistics.

3.2.3 Instructions

To meet our main goal of accessing network state, the ASIC must support read/write instructions that read values from the ASIC into packet memory, or write from packet memory into the ASIC. Besides read and write, a useful instruction in a concurrent programming environment is an atomic update instruction, such as a conditional store `CSTORE`, conditioned on a memory location matching a specified value.

To keep the ASIC design simple, we do not recommend instructions that change execution flow, with one exception. In our experience, we found the conditional execute (`CEXEC`) instruction to be very useful. For instance, it may be desirable to execute a network task only on one switch (see §2.2), or only on a subset of switches (say all the top of rack switches in a datacenter). The conditional execute instruction can be implemented by specifying a switch register, a mask, and a value, which instructs the ASIC to execute the TPP only when `(register & mask) == value`. Since instructions are executed sequentially, all instructions that follow a failed `CEXEC` check will not be executed.

3.3 Putting it together: the TCPU

We insert the TCPU just after the L2/L3/TCAM tables. Besides the specific TCPU processing, the ASIC forwards TPPs like regular packets; TPPs are therefore subject to congestion, or configured access control policies. Non-TPP packets are ignored by the TCPU.

The TCPU is a Reduced Instruction Set Computer (RISC) processor (Figure 5) that executes instructions in a five stage pipeline: (a) instruction fetch, (b) instruction decode, (c) execute, (d) memory read and (e) memory write. The header parser completes stage (a) by the time the packet reaches the TCPU, and stores the fixed size instructions in a temporary buffer. With read/write/simple arithmetic instructions, each stage takes only 1 cycle. Since instructions are pipelined, this RISC processor runs at a throughput of 1 instruction per clock cycle, with a latency of 4 cycles. Note that the entire packet passes through a series of registers before it is copied to memory, so all modifications to the packet are in local buffers. Finally, all packet modifications are commit-

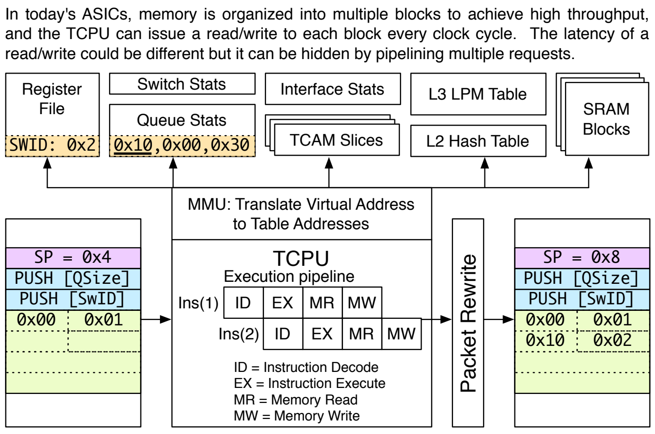


Figure 5: **A TCPU implementing the standard RISC processor pipeline supporting instructions in Table 1.**

ted to the packet before it is copied to switch memory (to save memory bandwidth).

Overheads. Our initial Linux prototype supports all instructions in Table 1, and we were able to encode an instruction and its operands in a 4-byte integer. If we limit to 5 instructions per packet, the instruction space overhead is 20 bytes/packet; if each instruction accesses 8-byte values in the packet, we require only 40 bytes of packet memory per hop. Low-latency ASICs today can switch minimum sized packets with a cut-through latency of 300ns [9], which is 300 clock cycles for a 1GHz ASIC. Since TPPs can be pipelined with other modules, we believe restricting a TPP to a handful of instructions makes it feasible to implement the RISC processor directly in the datapath, without incurring additional latency.

4. DISCUSSION

In the previous sections, we showed how TPPs enable end-hosts to access network state with low-latency, which can then act on this state to achieve a certain functionality. This is attractive as it makes the ASIC *future proof* (to an extent), making it possible to deploy interesting functionality at software-development timescales. We now discuss a few important concerns.

What about security? The very thought of packets querying and modifying network state induces worry to a network operator, even if the memory map (§3.2.1) isolates critical forwarding state from state modifiable by TPPs. We think this architecture makes sense (at least initially) in networking environments operated by a single entity (e.g., datacenters, or an enterprise network, or a single Autonomous System), where it is easy to exercise strict control over what packets are injected into the network. For example, in multi-tenant or untrusted environments such as public cloud datacenters, the ingress switches at the network edge (the virtual switch, or the border routers) can strip TPPs injected by VMs, or those TPPs received from the Internet.

How are TPPs different from other proposals to program a network? TPPs fit into a wide spectrum of dataplane programmability available today [10, 12, 13], and proposed in literature recently [14]. In the dataplane, today's ASICs support configurable packet matching using Ternary Content Addressable Memory, and programmable actions through custom 'field processors' [12, 14]. These action processors apply a sequence of actions (such as push headers, pop headers, rewrite fields) to a filtered subset of packets. Our work is complementary to the above switch-centric proposals and differs in two ways: (a) we focus on refactoring network-wide tasks to take advantage of the dataplane programmability, and (b) the TPPs are within the packets and hence can be changed rapidly, in contrast to TCAM rules that are updated slowly in the control plane.

There have been numerous efforts to expose switch statistics through the dataplane, particularly to improve congestion management and network monitoring. One example is Explicit Congestion Notification (ECN) in which a router stamps a bit in the IP header whenever the egress queue occupancy exceeds a configurable threshold. Another example is IP Record Route, an IP option that enables routers to insert the interface IP address on the packet. Instead of anticipating future requirements and designing specific solutions, we adopt a more generic approach to accessing switch state.

Parting remarks. Our goal in this paper is to enable end-hosts to flexibly measure and control network behavior. Our key insight is that providing end-hosts with low-latency access to shared network state helps in realizing this goal. To maintain flexibility, we designed a programmable interface to access network state. To achieve low-latency, we proposed tiny packet programs that query and modify network state directly in the dataplane. By identifying simple instructions that can execute at line-rate in hardware, we were able to push the responsibility of carrying out complex computation on network state to end-hosts. ASICs have long been the most cost-effective way of obtaining high bandwidth. Our design to split functionality between end-hosts and the network is largely driven by the key requirement to meet the stringent performance requirements of the ASIC.

Though read-write TPPs already enable a wide range of network tasks, we stress that TPPs have not been designed with completeness in mind to solve any and all networking tasks. Understanding the tradeoffs between implementing primitives within the network versus the end-hosts is an important direction for future work.

Acknowledgments

The authors would like to thank Brandon Heller, Kok-Kiong Yap, Sarang Dharmapurikar, Srinivas Narayana, and Patrick Bosshart for helpful feedback and discussions. The work at Stanford was funded by NSF FIA award CNS–1040190. Opinions, findings, and conclusions do not necessarily reflect the views of NSF or other sponsors.

References

[1] Nandita Dukkipati and Nick McKeown. Why Flow-Completion Time is the Right metric for Congestion Control. *ACM SIGCOMM CCR*, 2006.

[2] David L Tennenhouse and David J Wetherall. Towards an Active Network Architecture. *DARPA Active Networks Conference and Exposition*, 2002.

[3] Beverly Schwartz, Alden W Jackson, W Timothy Strayer, Wenyi Zhou, R Dennis Rockwell, and Craig Partridge. Smart packets for active networks. *Open Architectures and Network Programming Proceedings*, 1999.

[4] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990.

[5] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. *USENIX NSDI*, 2013.

[6] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *ACM SIGCOMM*, 2012.

[7] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. *ACM SIGCOMM*, 2012.

[8] Brandon Heller, Nikhil Handigol, Vimalkumar Jeyakumar, Nick McKeown, and David Mazières. Where is the debugger for my Software-Defined Network? *ACM HotSDN*, 2012.

[9] Intel Fulcrum FM4000 ASIC. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-switch-fm4000-datasheet.pdf`, Retrieved July 1, 2013.

[10] Intel Fulcrum FM6000 ASIC. `http://www.ethernetsummit.com/English/Collaterals/Proceedings/2013/20130404_S23_Ozdag.pdf`, Retrieved July 1, 2013.

[11] Multipath proposal for OpenFlow. `http://www.openflow.org/wk/index.php/Multipath_Proposal`, Retrieved July 1, 2013.

[12] Eric A Baden, Mohan Kalkunte, John J Dull, and Venkateshwar Buduma. Field processor for a network device, 2010. US Patent 7,787,471.

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 2008.

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM*, 2013.