Enabling Active Networking on RMT Hardware

Rajdeep Das and Alex C. Snoeren
UC San Diego

ABSTRACT

The ecosystem of application functionality built around programmable switch hardware is growing at a rapid pace in recent times, fueled by the advent of reconfigurable match-action table (RMT) technology. This new class of devices is capable of simultaneously delivering basic computation and line-rate forwarding at a reasonable cost. Given this transformation in commodity switching functionality, we suggest it may be time to reconsider the concept of active networking, where end hosts can off-load application functionality to the network in real time without requiring the assistance of the network operator. We present a preliminary approach to encoding (nearly) arbitrary computation into a series of network packets that can be decoded and executed on programmable switch hardware. Our programs can leverage both the forwarding and storage capabilities of RMT devices. We also conduct an initial exploration into the importance of dynamically allocating switch resources across active programs to improve aggregate performance.

ACM Reference Format:
Rajdeep Das and Alex C. Snoeren. 2020. Enabling Active Networking on RMT Hardware. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3422604.3425934

1 INTRODUCTION

Programmable switch hardware, such as that based on the reconfigurable match-action table (RMT) paradigm [4], has become increasingly popular in recent years. This emerging class of devices is capable of performing basic computation inside the network while forwarding packets at line rates. Researchers have proposed a wide variety of ways to leverage RMT-based programmable switching to offload functionality from end hosts, often taking particular advantage of the unique topological advantages afforded by in-switch processing. Recent systems demonstrate performance improvements in domains as varied as data aggregation [9], machine learning [10], object caching [7], distributed consensus [5] and network telemetry [6] among others. The sheer number of disparate target domains and velocity of evolution of the associated P4 [3] ecosystem suggest there remain many additional benefits that are yet untapped.

We observe, however, that the capabilities of the current, P4-based deployment model fall considerably short of earlier visions for in-network computation. In particular, P4 application logic is statically deployed by operators at network configuration time. Active networking [2, 11], in contrast, sought to enable user traffic to execute arbitrary programs inside the network, treating the switching fabric as a general-purpose execution environment. This more flexible approach provides the potential to dramatically expand the set of applications and users that can benefit from switch-based processing present in the network. Unfortunately, commercial switching hardware of the late 1990s and early 2000s was unable to provide line-rate forwarding performance while executing active programs, presenting a significant barrier to the adoption of proposed active-networking technologies. In subsequent years network processors were developed that are able to deliver line-rate network performance while allowing rich functionality, but their cost remains prohibitively high when compared to commodity switching gear. The increasing commoditization of RMT-based hardware, however, suggests price may no longer be a significant barrier.

We argue that the time is ripe for a reconsideration of active networking, in particular to study whether RMT hardware may finally unlock the promises of the original, traffic-directed computation vision. As a starting point, we show it is possible to use P4 to turn a commodity RMT switch (a Barefoot Tofino Wedge100BF-65X) into a something akin to a virtual machine, where program instructions contained within incoming network packets are executed while the packets are forwarded through the switch, potentially impacting not only the contents and forwarding behavior of the packet itself, but also the state of the switch—which, transitively, may impact the forwarding behavior experienced by subsequent packets. Each packet can therefore perform a piece of computation that contributes to the overall functionality of a corresponding off-loaded application. We leverage the storage capabilities of programmable RMT hardware to maintain application state across packets and deliver complex functionality impossible to describe within a single packet.

Our approach naturally enables multi-processing within the switch. Because each incoming packet can contain its own, independent program, a switch may execute instructions for multiple programs concurrently. Importantly, unlike the current P4 model, where the network operator must determine the set of applications to support in a given switch *a priori*, in our model a new application can be executed on a switch simply by sending traffic containing that program—the switch does not need to be reconfigured in any way. Obviously, our model raises all of the standard questions regarding resource isolation and scheduling, namely defining both mechanisms and policies for protection and sharing. We defer almost all of these to future work, instead choosing to focus on the pragmatic question of feasibility: specifically, is it possible to replicate the functionality of recently-proposed P4 programs in our model? And, critically, can multiple such programs execute concurrently on a single switch while retaining their inherent performance benefits? For the particular applications we start with, the latter question comes down to one of resource management—can we dynamically allocate on-switch memory resources among competing programs to achieve better overall application performance?

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11.
https://doi.org/10.1145/3422604.3425934

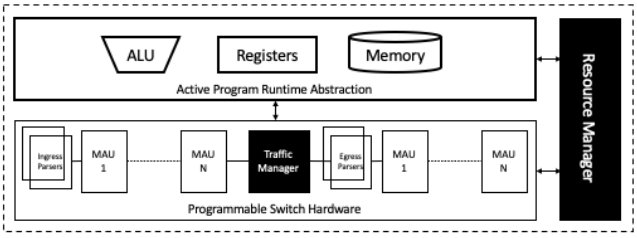

Figure 1: System model

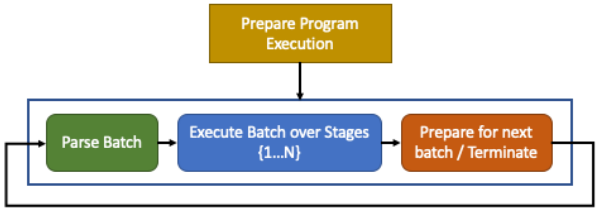

Figure 2: Program execution runtime

We describe the space of programs that can be expressed with our programming model and introduce two example programs selected from the set of applications that has been implemented using P4. We choose one commonly used network function—a load balancer—as well as an example of application offload: in-network caching. Finally, we demonstrate the importance of the memory manager that resides in the control plane of the switch. In particular, we illustrate the impact of initial mechanisms that guide and enforce memory management on the performance the network applications are able to achieve. We conclude with a discussion of the remaining technical challenges that must be addressed before active networking might finally become a reality.

2 PROGRAM EXECUTION

Our overall goal is to execute programs contained within network packets on the data plane of an RMT programmable switch. We do so by writing a P4 program that implements a runtime execution environment for a custom instruction set designed to exploit the functionality of RMT hardware. Figure 1 shows an overview of our system. Programs encoded inside packets are executed on a runtime deployed on the switch data plane. A resource manager that runs on the switch control plane enables multi-tenancy by isolating and limiting resources among applications. We start in this section by describing our instruction execution model, in particular how received instructions are processed by the switch hardware and the costs associated with executing them. We then turn in the next section to the programming model we use to encode functions inside network packets. Our initial instruction set is expressive enough to encode complex functions such as database joins while concise enough to implement in a single P4 program.

2.1 Packet flow

The P4 runtime seeks to maximize the switch resources available to the packet-based program while maintaining generality. Figure 2 illustrates the behavior of the runtime program which consists of approximately 3,000 lines of P4 code. In the terminology of P4 programs, the ingress pipeline (shown in gold in the top half of the diagram) prepares the active program for execution as it enters the switch, while the program instructions actually execute in the egress pipeline (shown as the enclosing blue box in the bottom portion of the diagram). Each instruction in the program is executed in a distinct match-action stage in the egress pipeline. Hence, the number of instructions that can be executed in a single pass through the egress pipeline is limited by the number of stages in the pipeline. For active programs that require more stages, the packet is 'recirculated' back into the switch's egress pipeline. This process is repeatable arbitrarily at a cost of network throughput as the same packet has to be processed more than once. Hence, some control-flow constructs in our instruction set such as jumps and loops incur a performance cost that must be taken into account when writing programs.

2.2 Instruction parsing

At the beginning of each pass through the egress pipeline, a batch of instructions is prepared to be executed in that pass (shown in green) and stored inside a special data structure specific to the programmable switch hardware. These instructions are parsed from the network packet by means of the switch's programmable parser and stored in an internal data structure known as the packet header vector (PHV). Each batch extracts the maximum number of instructions that can be executed in the following set of match-action stages in the egress pipeline.

Packet parsing begins with the standard protocol headers including Ethernet, IP and UDP. A special header is then parsed which contains fields that store state and metadata information about the program. This includes a program identifier and an accumulator variable. The identifier is a 16-bit field which helps to distinguish between packets from different programs that are processed on the switch at the same time. (Various security policies could be implemented by keeping a list of 'verified' program IDs that are allowed to execute on a given switch, and requiring the ID to be a deterministic cryptographic hash of the active code, but we defer such considerations to future work.) The accumulator variable stores the output of the execution of the program, if any.

The next set of headers contains instructions corresponding to the active program. Each instruction contains an opcode, an 8-bit `label` described in Section 3 and a single, 16-bit argument (along with a set of bitflags described later). The programmable parser extracts a batch of instructions one at a time until a special `EOF` instruction is reached, which specifies the end of the program and terminates parsing. Our prototype uses 8-bit opcodes which is currently sufficient to encode all the instructions used in our programs. The flag field is currently 8-bits wide, resulting in a five-byte instruction format. With a standard MTU-sized packet of 1,500 bytes (and taking into account the other standard headers) this implies that programs of around 300 instructions can fit inside a single packet. This has been more than sufficient for all the programs that we have implemented using our model so far—including database joins—but programs can span multiple packets; they simply need to preserve intermediate state in switch memory as described below.

The parsing phase is followed by a series of match-action stages as consistent with the RMT architecture. The standard P4 compiler

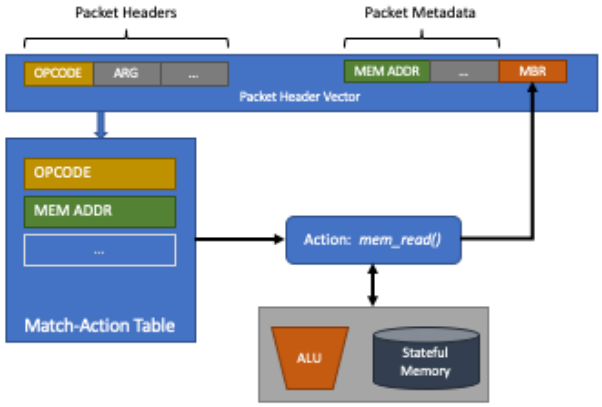

Figure 3: An example of executing a `MEM_READ` instruction; the vale stored at `MEM ADDR` in the switch's stateful memory is loaded into the `MBR` register.

exports a set of primitives which we use to provide ALU and other operations in our instruction set architecture. In each stage, the switch SRAM is allocated as one big P4 register array and exported as random-access-memory with read, write and counter semantics (as described below). As these registers are accessible by the switch control plane, we leverage this to perform memory management.

3 PROGRAMMING MODEL

Our active programming model is built on top of the semantics exported by the P4 programming language and the associated RMT architecture. An active program consists of a sequence of instructions contained within one or more packets. Execution follows the von Neumann model, where instructions are executed sequentially: once an instruction is executed by any switch in the network it will not be executed again. In our initial work, we consider programs that run to completion inside of a single switch (possibly taking multiple passes through the egress pipeline) but we imagine it may be preferable for packets to execute across switches in networks with multiple RMT-capable devices.

3.1 Instruction processing

Figure 3 shows the relationship between match-action tables, packet and ephemeral metadata contents (stored inside the PHV) and switch storage and computational units. Arithmetic, logical and bit-shift operations are enabled by the action units in the RMT pipeline. Instructions and their arguments are decoded by the match tables and executed by the corresponding action units. The match tables use the opcode (loaded by the parser) to determine which action to execute. Space prohibits us from listing a complete set of operations, but we illustrate several in the examples that follow.

The runtime uses a variety of packet metadata buffers to facilitate the control flow of the program. Data flows between packet header fields and these metadata buffers during program execution. The instruction set architecture exposes these buffers as a fixed set of general-purpose 'registers' that are available for use by operations in the instruction set. These registers can be loaded from and stored to both packet header fields and persistent memory on the switch through a set of memory access instructions. Two registers of particular interest are the memory address register (`MAR`) and memory buffer register (`MBR`), so named to reflect the semantics of the corresponding registers in a general-purpose CPU. The instruction argument is used to pass information from the packet directly to the action unit. The `label` is also loaded by the parser and is used to mark locations in the program to which it may jump.

3.2 Memory model

Persistent switch memory is somewhat different than typical architectures and comes in two forms: directly addressable, and associative collision chains whose length is dictated by the number of match-action stages in the switch hardware. Direct memory locations are specified by 16-bit addresses, while values stored in collision-chain memory are accessed through 16-bit-wide keys.

In contrast to registers which are available throughout a program's execution, the persistent memory locations available to an instruction depend upon the hardware stage on which the instruction is executed: both memory addresses and keys are scoped by the current pipeline stage. Said another way, the location of an instruction in a program dictates which portions of memory it can access, as memory physically located at one match-action stage cannot be accessed from another stage. This RMT-based restriction currently necessitates some careful program construction, but could potentially be alleviated by clever compiler design: Memory in later stages can always be accessed by inserting the necessary number of `NOP` instructions before the access, while locations on previous stages require re-circulation back into the egress pipeline to execute the instruction at the appropriate stage in the next pass.

Memory protection is enforced by the runtime during program execution. Allocation policies defined in the control plane allocate regions for corresponding applications. Memory re-allocation is triggered whenever the set of tenant applications change. An incorrect memory access by an instructions results in the generation of a segmentation fault, where the corresponding active packet is returned to the sender with a specific bit set in the active program header. The application then retrieves the allocated memory region from the switch runtime.

3.3 Control flow

Our current instruction set supports both branching and a basic loop construct. Apart from loops, our runtime currently cannot re-execute instructions already executed previously. In other words, one can only jump forward in the program. This can of course be partially compensated for by repeating instructions as necessary, but a practical implementation would likely remove this restriction.

3.3.1 Branching. A jump instruction branches to label in a different part of the program while a return instruction terminates execution of the program and branches out of the program. `UJUMP` is unconditional, while `CJUMP` checks the value of `MBR` and jumps if the value is not zero, while `CJUMPI` jumps if the value is zero. Unlike typical execution environments, however, active programs have constant-time execution despite the presence of conditional branches. All branches of a conditional are visited irrespective of the condition. Instructions in inactive branches are simply ignored (i.e., treated as `NOP`) while the active ones are executed. Return instructions are used to terminate execution of the program. This

```
1. MAR_LOAD, 0x0001
2. MEM_READ
3. CJUMP, goto=2
4. MBR_ADD, 1, label=2
5. MEM_WRITE
6. RETURN
```

Listing 1: Conditional example

```
1. MBR_LOAD, 3
2. LOOP_INIT
3. DO, goto=2
4. MBR_SUBTRACT, 1
5. WHILE
6. NOP, label=2
7. RETURN
```

Listing 2: Loop example

is a mandatory instruction for every program. A regular RETURN instruction marks the program execution as complete immediately and can be invoked anywhere in the program. The CRET instruction returns only if MBR is non-zero.

We illustrate both memory access and branching using a trivial example shown in Listing 1 that increments a value stored in memory location 0x0001 only if the value is currently zero, and stores the result in a different location. The first instruction sets the value of MAR to the address literal 0x0001. The MEM_READ instruction then loads the value at that memory location (in stage two of the pipeline) into MBR. The CJUMP instruction causes the next instruction (number 4, labeled 2) to be skipped if the value of MBR is non-zero. Else, it executes the MBR_ADD instruction which adds the numeric literal 1 to the contents of MBR. Finally, the MEM_WRITE instruction stores the value in MBR back to the address specified in MAR—but this time into the memory bank of stage five.

3.3.2 Loops. A loop follows the 'do-while' idiom common in many higher-level programming languages. We explain how a loop is written using the example shown in Listing 2. The loop conditional that determines whether the next iteration of the loop should be executed or not is based on the value in MBR. A non-zero value indicates that the loop should continue. In the example MBR is loaded with a value of 3 to cause the loop to execute three times. Once this value reaches 0 the loop will terminate. The LOOP_INIT is used to initialize the loop by indicating that the next instructions are iterable. The DO instruction implements the loop conditional and checks whether the value in MBR is zero or not. The goto label specifies where to branch to if the loop is done. The target location in the program is specified by a label. In the example if MBR is zero then it jumps to the instruction labeled 2 (i.e., instruction six). Else, the body of the loop is executed. In this case the following MBR_SUBTRACT instruction is executed which decrements the value in MBR by 1. The WHILE instruction is the final instruction in the loop construct and indicates the end of the scope of the loop. Control-flow instructions such as the branches described in the previous section can also break out of the body of a loop.

```
1. LOAD_5TUPLE
2. HASH_GENERIC
3. RANDOM_PORT
4. CMEM_WRITE
5. CMEM_WRITE
6. CMEM_WRITE
7. CMEM_WRITE
8. C_ENABLE_EXEC
9. SET_PORT
10. RETURN
```

Listing 3: Program for toy stateful load balancer

3.4 Examples

We have used our programming model to implement a variety of network functions. Here, we present two toy examples based upon published applications implemented on RMT hardware using P4 to demonstrate our ability to express similar functionality in our framework. Then, in the subsequent section, we consider performance concerns raised by our model.

3.4.1 Stateful load balancer. Stateful flow-level load balancers have been implemented in the P4 ecosystem before [1] and require forwarding packets based on state. We wrote an implementation of a trivial load balancer using our active programming language that selects a random egress port to use for the duration of a flow; the program itself need only be included in the first packet of the flow—or whenever the network chooses to redirect the flow. (A useful load-balancing application, of course, is likely to select only among a subset of the egress ports, and may need to adjust the packet destination addresses while doing so.)

Listing 3 shows the function written using our active language. The instruction LOAD_5TUPLE loads the packet's 5-tuple (i.e., source and destination IP address and port, along with the IP next protocol field) into a data structure. The subsequent instruction HASH_GENERIC applies a hash function to the contents of this structure and stores the result in MAR. The instruction RANDOM_PORT selects a random output port and stores it in MBR. The SET_PORT instruction at step 9 updates the switch's forwarding table to set the egress port for all packets associated with this packet's 5-tuple (i.e., the current flow) to the contents of MBR (i.e., the result of the hash). The sequence of associative MEM_WRITE instructions in steps 4–8 attempt to write the contents of MBR to the memory location associated with the key stored in the MAR in each of the four stages following the semantics of a collision chain. (In particular, any successful MEM_WRITE will prevent the execution of any subsequent ones.) The C_ENABLE_EXEC instruction enables the subsequent (SET_PORT) instruction only if the associative write was successful. Packets that are unable to store their selected output port (due to an overflowed collision chain) are dropped in this trivial example.

3.4.2 Object cache. We draw inspiration from systems like NetCache [7] to implement an in-network cache using our prototype language. A request for an object received at the switch is first looked up in its local storage. If it is found then the object value is returned to the requester, else the switch forwards the packet to its

```
1.  MAR_LOAD, <key>
2.  CMEM_READ
3.  CMEM_READ
4.  CMEM_READ
5.  CMEM_READ
6.  ENABLE_EXEC
7.  CJUMPI, 2
8.  RTS
9.  ACC_LOAD, label=2
10. RETURN
```

Listing 4: Program for toy cache read

intended destination as usual. The cache uses the stateful memory on the switch to implement a key-value cache. Values are accessed in a collision chain across several stages. Listing 4 shows a program for reading objects from the switch cache. (Storing objects requires a separate, complimentary program which we do not include here due to space constraints.)

The first instruction loads the 16-bit key (specified in the packet as a literal instruction argument) into `MAR`. The next four instructions attempt to read the value associated with `MAR` using the semantics of a collision chain. (Any successful `CMEM_READ` instruction disables the execution of all subsequent ones.) The `ENABLE_EXEC` instruction is used to resume execution of the program once it is paused due to a successful memory access. The `CJUMPI` instruction performs a conditional jump if a cache hit occurs and executes the `RTS` instruction. This instruction marks the packet for returning back to the sender. The subsequent `ACC_LOAD` instruction is executed thereafter and the value of the `MBR` (which contains the object value) is loaded into the packet header. Otherwise, a cache miss occurs and the packet is forwarded along to its destination—presumably a server which can service the request.

4 MANAGING SWITCH RESOURCES

One of the key challenges raised by active networking is determining how to multiplex scarce switch resources across programs, not all of which may be executing at a given time (e.g., some programs may require continued use of stable switch storage even when none of the application's packets are transiting the switch). In particular, the performance of many of the applications that have been proposed for off-loading depends greatly on the resources available to the application at the switch. Here, we consider two of these resources: bandwidth and memory.

4.1 Limiting recirculation

While a 64-port switch with 100-G ports has enough backplane capacity to process 6.4 Tbps of traffic, packet recirculation changes things completely. A few recirculations can have significant effects on forwarding performance due to both available processing capacity and queue space. Larger and more complicated active programs are likely to require multiple passes through the switch and, hence, recirculation. However, as datacenter networks frequently have a surfeit of switching capacity it may make sense to transform this surplus bandwidth into useful computation. We attempt to manage

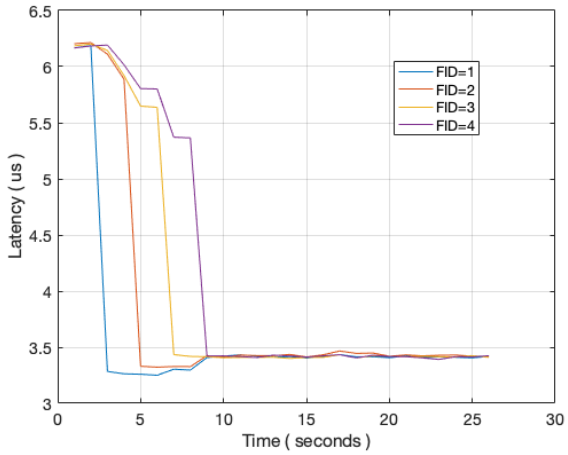

Figure 4: A demonstration of dynamic memory allocation. Initially, all four flows bypass the switch cache. Then, each flow begins using a separate active caching application that attempts to serve requests from on-switch memory. As more active applications arrive, per-application allocations decrease, resulting in a higher miss rate and increasing latency.

this tradeoff in our prototype by implementing a recirculation manager for active programs. Our runtime monitors traffic levels on the switch using traffic meters based on the standard three-color marking scheme. Significant changes in traffic levels are reported to an application running on the switch control plane, which uses this information to dynamically assign limits to the number of allowed recirculations for active programs.

4.2 Memory allocation

Unlike bandwidth, which may be readily available in many environments, memory in the form of SRAM is likely to be scarce on most switches. While a single application using the switch can liberally use all the memory available on the device, careful partitioning of the memory space is required to enable multi-tenancy. Since the set of applications using the switch resources vary over time, memory management needs to be dynamic and automated based on predefined policies. In our prototype implementation of the memory manager, we explore a strawman approach of allocating memory equally to applications sharing the switch. Memory is explicitly requested by each new application, following which the memory manager reassigns memory equally among the new set of applications. For applications that benefit equally from memory, this approach enables fair sharing of resources.

To evaluate this hypothesis, we performed an experiment where we used four, separate caching applications (similar to the one we described in Section 3.4.2, but more full-featured) that share the switch over time. The keys corresponding to the requests are drawn from a Zipf distribution with $\alpha = 2$. All four applications first avoid the switch cache and directly access objects from the back-end key-value store. Then, each of these applications leverage the switch cache, one by one, over time. Figure 4 shows a timeline of the response latencies of application-level requests, averaged over one second. In the absence of on-switch caching, the destination server is able to respond to the requests in just over 6 microseconds.

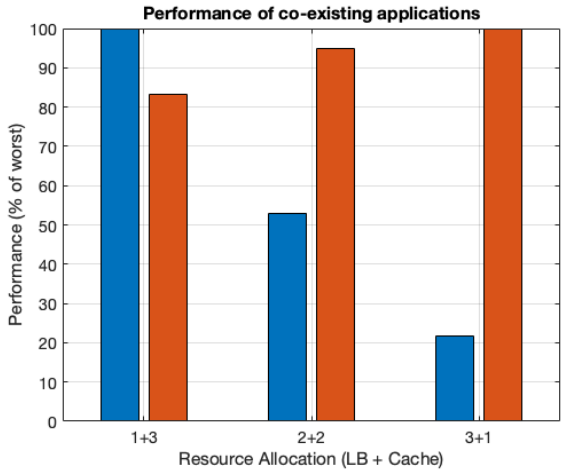

Figure 5: Performance benefits of memory allocation. We allocate four switch stages' worth of memory across two applications in varying proportion. The normalized load balancer drop rate is shown in blue, and response latency in orange.

As we can see, the latencies for each application flow converge over time as the memory manager equally allocates memory among them—with latencies increasing slightly as each arrives.

Of course, the performance impact of decreased resource allocations varies across applications. Figure 5 shows the relative performance benefits of allocating varying number of stages to both the load balancing application (from Section 3.4.1, in blue in the figure) and the cache application (shown in orange). In this experiment, both applications are running concurrently (i.e., the incoming traffic consists of packets carrying object requests and flows to be load balanced), and we consider three different allocations of four switch stages worth of memory. The left-hand grouping shows when one stage is allocated to the load balancer and the remaining three to the cache, the center evenly across applications, and the right-hand bars consider the reverse. The bars plot an application-specific performance metric (request latency in the case of the cache and flow drop rate in the case of the load balancer) normalized to the worst-case performance where each application is allocated only one stage. While the absolute heights of the bars are incomparable across applications, in both cases lower is better. One can deduce that the load balancer extracts greater relative performance benefit than the cache from a commensurate memory allocation.

5 OPEN CHALLENGES

Our initial exploration focuses exclusively on determining whether it is worthwhile from a performance perspective to reconsider active networking given today's hardware capabilities. Many challenges to running arbitrary code embedded in network packets remain, most notably the serious security implications. Previous proposals such as signing the active program and using the control plane to make the network aware of accepted signatures could still apply, but there have been recent advances on this front as well. Moreover, the fixed-function capabilities (such as hashing) offered by programmable switches may help in restricting the hazards of untrusted code.

In addition to security and performance implications, we also need to consider the pragmatism associated with deploying an active network. One such consideration is network goodput. Since active programs can occupy a significant fraction of a network packet payload, this effectively reduces the goodput of the network. This payload does not contain information that is consumed by the end-hosts and is hence an overhead to the network. One way of solving this problem would be to cache the active program on the switches to the extent possible. Similar mechanisms were proposed previously [11] where a function identifier could be used to retrieve code stored on the forwarding device; the capabilities of current programmable switches can further optimize such an approach.

Our resource management policies are preliminary. Applications such as caches will benefit less from dynamic reallocation of resources than applications whose performance scales linearly with memory such as load balancers. Policies need to be weighted according to the impact they have on application and network performance. For example, longer active programs that require multiple recirculations (such as our object cache) consume more switch-plane capacity. While active network programmers need to keep this in mind while writing programs, some assistance from the runtime may help minimize the potential negative impacts on both network as well as application performance.

Some of the robust and comprehensive P4 implementations of our toy applications such as NetCache and SilkRoad spread functionality over both the switch control and data planes. Because our current system does not allow active applications to leverage the capabilities of a switch's control plane our active applications are forced to replace the control plane functionality with additional functionality in the dataplane. While our initial experience suggests this tradeoff is possible, it may not always be performant. Hence, we are exploring ways to enable active programs to dictate control-plane functionality.

Keeping this in mind, we believe that the large emerging space of application functions that are being built on top of the P4 ecosystem makes it worth reconsidering the way these functions are executed on a P4 switch. It may make sense for network behavior to be directly controlled by end-host applications without having to interact with network administrators. Recent approaches like INT [8] aim to achieve this for specific use cases such as telemetry. Further, service changing in virtual network functions also seems likely to benefit from end-host control. Our approach attempts to deliver the flexibility to meet all of these needs using ideas from active networking introduced decades ago, but applied to modern programmable switch hardware. As recently pointed out [12] by researchers who originated active networking, the ecosystem built around programmable switches has enabled achieving something that was frequently perceived as lacking utility earlier.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (through grants CNS-1564185, CNS-1629973, and CNS-1911104) and the Advanced Research Projects Agency (ARPA-E). We thank the anonymous reviews for their comments on an earlier draft of this manuscript, and the Barefoot FASTER community for their assistance with P4 programming.

REFERENCES

[1] T. Barbette, C. Tang, H. Yao, D. KostiÄĞ, G. Q. M. Jr, P. Papadimitratos, and M. Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. pages 667–683, 2020.

[2] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Active networking and the end-to-end argument. In *Proceedings 1997 International Conference on Network Protocols*, pages 220–228, Oct. 1997.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[4] 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 *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM. event-place: Hong Kong, China.

[5] H. T. Dang, M. Canini, F. Pedone, and R. SoulÃĺ. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.

[6] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. ACM. event-place: Budapest, Hungary.

[7] X. Jin, X. Li, H. Zhang, R. SoulÃĺ, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM. event-place: Shanghai, China.

[8] C. Kim, A. Sivaraman, N. P. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band Network Telemetry via Programmable Dataplanes. 2015.

[9] A. Lerner, R. Hussein, and P. Cudre-Mauroux. The Case for Network-Accelerated Query Processing. page 10.

[10] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. RichtÃąrik. Scaling Distributed Machine Learning with In-Network Aggregation. *arXiv:1903.06701 [cs, stat]*, Feb. 2019. arXiv: 1903.06701.

[11] D. Wetherall. Active Network Vision and Reality: Lessions from a Capsule-based System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 64–79, New York, NY, USA, 1999. ACM. event-place: Charleston, South Carolina, USA.

[12] D. Wetherall and D. Tennenhouse. Retrospective on "towards an active network architecture". *ACM SIGCOMM Computer Communication Review*, 49(5):86–89, Nov. 2019.