Event-Driven Packet Processing

Stephen Ibanez
Stanford University

Gianni Antichi
Queen Mary University of London

Gordon Brebner
Xilinx Labs

Nick McKeown
Stanford University

ABSTRACT

The rise of programmable network devices and the P4 programming language has sparked an interest in developing new applications for packet processing data planes. Current data-plane programming models allow developers to express packet processing on a synchronous packet-by-packet basis, motivated by the goal of line rate processing in feed-forward pipelines. But some important data-plane operations do not naturally fit into this programming model. Sometimes we want to perform periodic tasks, or update the same state variables multiple times, or base a decision on state sitting at a different pipeline stage. While a P4-programmable device might contain special features to handle these tasks, such as packet generators and recirculation paths, there is currently no clean and consistent way to expose them to P4 programmers. We therefore propose a common, general way to express *event processing* using the P4 language, beyond just processing packet arrival and departure events. We believe that this more general notion of event processing can be supported without sacrificing line rate packet processing and we have developed a prototype event-driven architecture on the NetFPGA SUME platform to serve as an initial proof of concept.

ACM Reference Format:
Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. Event-Driven Packet Processing. In *The 18th ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3365609.3365848

1 INTRODUCTION

Programmable network devices have been gaining significant traction within the networking community as a result of their unique ability to deploy custom algorithms that operate at line rate. There have already been many interesting applications that take advantage of this new found ability to program the data plane [6, 10, 12, 13, 18]. P4 has emerged as the *de facto* language for programming the data plane. P4 programs are designed to be compiled onto a class of data-plane architectures called Protocol Independent Switch Architecture (PISA) [2]. PISA architectures are composed of programmable parsers, match-action pipelines, and deparsers and are designed to process packets at line rate. Each instance of a PISA architecture exposes a certain data-plane programming model to the P4 programmer who then works within the confines of the provided programming model to implement their custom processing logic. Every data-plane programming model is driven by a set of data-plane events, where a data-plane event is an architectural state change that triggers processing in the programming model.

The simple PISA architecture introduced in [2] consists of a single programmable parser, match-action pipeline, and deparser connected in series. The P4 language consortium recently defined a different PISA architecture called the Portable Switch Architecture (PSA), which is depicted in Figure 1. The PSA consists of two P4 programmable pipelines, one to process packets on ingress and one to process packets on egress as they leave the device. Both of these architectures are what we call *baseline PISA* architectures. A baseline PISA architecture supports a programming model that exposes synchronous packet-by-packet processing to the P4 programmer. That is, the programming model only allows developers to define how to handle a small set of packet-related events, usually ingress and egress packet events.

We observe that many data-plane algorithms do not naturally fit into this synchronous packet-by-packet programming model. Some applications need to execute logic independently of packet arrivals and departures. For example, HULA [14] is a load balancing application that must periodically generate probe packets to measure link utilization. When deployed on a baseline PISA architecture, these HULA probe packets must be generated by either the control plane or end hosts because the programming model provides no means to perform periodic tasks or generate packets. Similarly, the Count-Min Sketch (CMS) [5] is a commonly used data-plane primitive that must be periodically reset. When a CMS is used in a baseline PISA architecture, the control

HotNets '19, November 13–15, 2019, Princeton, NJ, USA

ACM ISBN 978-1-4503-7020-2/19/11…$15.00
https://doi.org/10.1145/3365609.3365848

plane must be responsible for performing the reset operation. This can lead to significant overhead for the control plane, especially if the data structure must be frequently reset. Other data-plane operations, such as measuring flow rates or computing average queue occupancies, must compute functions of a signal over a moving window of time. While this type of operation is sometimes possible to implement using only packet events, it is often cumbersome and challenging to do so. Furthermore, many data-plane applications can benefit from the ability to update algorithmic state multiple times while processing a packet. For instance, computing congestion signals such the number of buffered flows, inherently require state updates both as packets are enqueued and dequeued from the buffer.

In this paper we introduce *event-driven PISA* architectures, which provide a programming model that explicitly exposes a rich set data-plane events to the P4 programmer. As packets traverse the architecture, they generate events such as buffer enqueue, dequeue, or overflow events, which are subsequently handled by dedicated processing threads that share state with the packet processing threads. Events may also be generated independently of any packet processing logic, such as based on a timer configuration, a link status change, or a control-plane command. An event-driven programming model allows a P4 programmer to express how each of the individual data-plane events are handled. Event-driven PISA architectures alleviate many of the key limitations of baseline PISA architectures. In particular, they enable more interesting stateful packet processing applications as they allow data-plane programs to spawn threads that perform background maintenance of algorithmic state, as well as perform periodic tasks such as generating packets.

The main contributions of this paper are:

- We propose a common, general way to express line rate data-plane *event processing* beyond just packet arrival and departure events.
- We identify a set of useful data-plane events that can be used to implement a wide range of data-plane algorithms.
- We identify classes of applications that will benefit from the proposed programming model.
- We demonstrate feasibility of the approach at line rate by architecting an event-driven architecture on the NetFPGA SUME platform.

2 EVENT-DRIVEN PROGRAMMING

P4 programs are often compiled to run on a PISA pipeline comprising multiple *match+action* stages. The baseline PISA architecture only supports events that are triggered by packet arrivals and departures. In this paper we will explore how

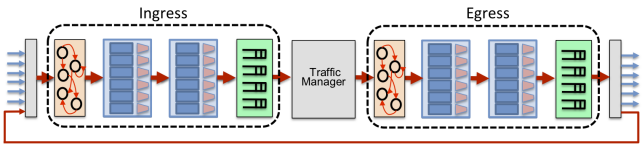

Figure 1: Simplified diagram of the portable switch architecture (PSA), which consists of separate ingress and egress pipelines to handle packet arrival and departure events, respectively.

to enhance the baseline programming model to support a richer set of data-plane events.

We started by examining a broad set of data-plane applications in order to identify a set of useful data-plane events. Our list is shown in Table 1. The first three are packet events (ingress, egress, and recirculated) and are commonly supported in the baseline programming model. The remaining events, such as when a packet is enqueued or dequeued, a buffer overflows or underflows, a timer expires, or a link status changes, are sometimes available from the hardware, but are not exposed by the programming model.

Our event-driven PISA programming model explicitly exposes data-plane events to the P4 programmer by allowing them to define custom event handling logic. A particular target device exposes the precise set of events that it supports via the P4 architecture description file. This generalization from packet events to data-plane events gives data-plane programs much more flexibility, and if designed appropriately, still allows packets to be processed at line rate. We next describe how these events are exposed to data-plane developers within our proposed event-driven programming model.

Example Logical Architecture Model. We consider a simple event-driven architecture that only supports ingress packet events, enqueue events, and dequeue events. Figure 2 depicts a block diagram of this logical architecture model. Ingress packet events trigger processing in the ingress PISA pipeline. Every time a packet is enqueued in the switch buffer, the traffic manager extracts some metadata from the packet and uses it to fire an enqueue event which then triggers the

Table 1: Set of useful data-plane events to support in an event-driven packet processing architecture.

Data-Plane Events	
Ingress Packet	Buffer Overflow
Egress Packet	Buffer Underflow
Recirculated Packet	Timer Expiration
Generated Packet	Control-Plane Triggered
Packet Transmitted	Link Status Change
Buffer Enqueue	User Event
Buffer Dequeue	

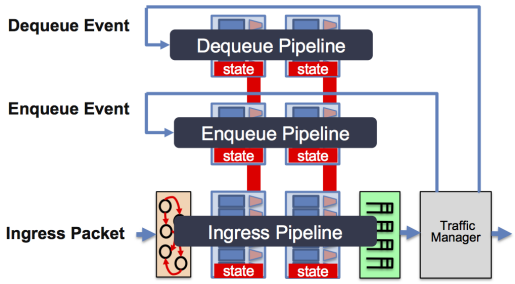

Figure 2: A diagram of a logical event-driven data-plane architecture. Each event triggers processing in a separate logical pipeline.

logical enqueue pipeline. A similar procedure occurs for dequeue events. Each of these pipelines have some notion of local state as well as global shared state.

Writing Event-Driven Programs. Let us see how we can write an event driven P4 program for our example architecture model. Our P4 program will monitor the buffer occupancy of each active flow. It will use this information to identify microburst culprits: flows that contribute to a sudden, significant increase in buffer usage. As noted in [3], this is very challenging to do on baseline PISA architectures because the programming model does not allow state to be updated both when packets are enqueued and dequeued from the buffer. The authors needed to keep track of multiple, complex, stateful data structures to keep track of the (approximate) queue occupancy in the egress pipeline. If instead we use our event-driven programming model, we can reduce the stateful requirements at least four-fold and can perform the detection in the ingress pipeline *before* packets are enqueued in the switch buffer.

To do this, our P4 target architecture will need to support events as well as a new type of extern. An extern is an element whose functionality is not described in P4, and provides an interface for P4 programs to interact with it. This is the mechanism through which an architecture can expose stateful operations to P4 programmers. Our target event-driven architecture will support a new type of extern called `shared_register` to allow event processing threads to share state.

The user's program `microburst.p4` (shown below), instantiates one of these new extern objects to track the buffer occupancy on a per-flow basis (`flowBufSize_reg`). The `flowBufSize_reg` should be allocated with enough entries to track state for every flow that has at least one packet in the buffer.[1]

When a packet arrives, the ingress logic computes the packet's flow ID by hashing the IP source and destination addresses, and initializes the packet's metadata so that it can carry the enqueue and dequeue events through the pipeline. Next, the ingress logic reads the flow's buffer occupancy and checks if it exceeds a pre-configured threshold to determine if the flow is a microburst culprit. Upon successful detection of a microburst culprit, the program may then decide to take corrective action such as dropping the packet, lowering its scheduling priority, or notifying a controller.

```
// microburst.p4
shared_register<bit<32>>(NUM_REGS) bufSize_reg;

// Ingress Packet Event Logic
control Ingress(/* hdrs and metadata */) {
  bit<32> bufSize;
  bit<32> flowID;
  apply {
    // compute flowID
    hash(hdr.ip.src ++ hdr.ip.dst, flowID);
    // initialize enq & deq metadata for this pkt
    enq_meta.flowID = flowID;
    enq_meta.pkt_len = meta.pkt_len;
    deq_meta.flowID = flowID;
    deq_meta.pkt_len = meta.pkt_len;
    // read buffer occupancy of this flow
    bufSize_reg.read(flowID, bufSize);
    // detect microburst
    if (bufSize > FLOW_THRESH) { /* microburst
    culprit! */ }
  }
}
```

The user also needs to implement event handling logic for enqueue and dequeue events to update the per-flow buffer occupancy state. The enqueue logic increments the appropriate entry in the `bufSize_reg` by the length of the enqueued packet, while the dequeue logic decrements this state by the length of the packet that was just removed from the buffer. Here we show how the enqueue event handling logic can be implemented, the dequeue event handling logic being very similar.

```
// Enqueue Event Logic
control Enqueue(inout enq_data_t meta) {
  bit<32> bufSize;
  apply {
    // increment buffer occupancy of this flow
    bufSize_reg.read(meta.flowID, bufSize);
    bufSize = bufSize + meta.pkt_len;
    bufSize_reg.write(meta.flowID, bufSize);
  }
}
```

Perhaps the biggest benefit to the P4 programmer is that event handling logic can now be expressed in separate threads of execution with shared state. The issues surrounding such state are considered in Section 4.

3 EVENT-DRIVEN APPLICATIONS

Table 2 summarizes five classes of applications that we believe will greatly benefit from event-driven programming:

Congestion Aware Forwarding applications base their forwarding decisions on recent congestion signals. We can derive congestion signals, such as (per-active-flow) queue occupancy, link utilization, and packet loss from the enqueue,

[1] If needed, a count-min-sketch data structure can be used to reduce state requirements even further.

dequeue, and buffer overflow events. This allows for variants of ECN marking, with packets carrying multiple bits rather than just one, to communicate queue occupancy along the path, or just the maximum queue occupancy at the bottleneck. If the programmer uses timer events as well, congestion signals can be periodically transmitted along various paths in the network, as is the case in HULA [14] or can be used in the ingress pipeline to make priority forwarding decisions, as in NDP [8].

Network Management encompasses a broad range of tasks typically handled by the network control plane. For example, re-routing traffic when links fail usually requires the control plane to detect the failure, re-route the affected flows, and potentially migrate data-plane state from a flow's old path to its new one. By introducing link status change events, the data plane can immediately respond to link failures, autonomously re-route affected flows and migrate data-plane state. This makes it much easier to implement Fast Re-Route (FRR) [25] and swing-state [17]. Furthermore, timer events allow data-planes to reliably and quickly probe and detect failed neighbors and tunnels.

Network Monitoring with extremely fine-grain measurements made possible by In-band Network Telemetry (INT) [20] is becoming increasingly popular. One challenge with INT is the potentially huge volume of measurement data, which might overwhelm a software-based logging and analysis system. But if we can expose event-driven programming to the programmer, data-plane applications can analyze, preprocess and reduce the amount of data reports, using filters and watchlists. For example, data planes can use timer events to aggregate congestion information (e.g. queue size, packet loss, or active flow count) and only report anomalous events to the monitoring system periodically. Furthermore, given it is now easy to write programs using enqueue and dequeue events, applications such as microburst detection are now much simpler to write than before [3].

Traffic Management and packet scheduling are not currently supported in the P4 language. But an event-driven programming model helps to enable programmability of three major traffic management functions: active queue management (AQM), policing, and packet scheduling. AQM algorithms, such as RED [7], AFD [22], FRED [16], and PIE [23], need to monitor and manage the packet queues, and need access to several congestion signals in the ingress pipeline. These include: current queue occupancy, queue service rate, queueing delay, packet loss volume, rate of change of the queue size, per-active-flow queue occupancy, and number of active flows. Event-driven programming gives the user access to all of these congestion signals (and more). Thus, AQM is a natural use case of this approach, and was one of the motivating applications for our work. Similarly, policing often requires a leaky token bucket meter [9]. While baseline PISA architectures might expose fixed-function meters to P4 programmers as primitive elements [21], if we use timer events, token bucket meters can be constructed from simple registers. This approach allows data-plane developers to build and customize their own policing algorithms. Taking this one step further, we can construct a complete, programmable packet scheduler using our event-driven model in combination with the recently proposed Push-In-First-Out (PIFO) queue [27].

In-Network Computing became a hot topic once it was realized that programmable data planes can be used to accelerate some end-host applications. For example, NetCache [13] demonstrated improvements in throughput and tail-latency of key-value storage systems by caching hot items within a P4-based data-plane. Timer events allows the programmer to write more sophisticated cache replacement policies, such as approximate least-recently-used (LRU), entirely in the data-plane. Timer events can also be used to quickly clear all NetCache statistics, which, as the authors point out, would allow the cache to more rapidly react to workload changes. Link status change events enable coordination services, such as NetChain [12], to quickly react to network failures.

Overall, we have found that P4 programs for a wide range of applications can be simplified using the event-driven model. We conclude (perhaps unsurprisingly with hindsight) that *network algorithms are inherently event-driven.*

4 GLOBAL VS DISTRIBUTED STATE

One of the most important design decisions, when building an event-driven data plane, is how state is shared (or not) among different processing elements. If state is private and local to a pipeline stage, we need a way to share state between stages, potentially maintaining multiple copies. Things gets more complicated when a device has multiple independent pipelines (e.g. Tofino has four independent pipelines). Deciding how state is shared turns out to be a key design decision.

The answer depends on the line rate. Lower line rate devices (e.g. a WiFi AP) can use multi-ported memory to directly implement the logical event processing pipelines described in Section 2 as separate physical pipelines, each with a dedicated read/write port to global shared state. The memory would need as many ports as the number of event processing threads that access the state.

For high line rate devices, where multi-ported memory is impractical to implement, we require a different approach. For these devices, we can merge the logically separate event processing pipelines into a single physical pipeline so that

Table 2: Various application classes that can benefit from event-driven programming.

Application Classes	Examples	Events Used
Congestion Aware Forwarding	Load Balancing [1, 14], Congestion Control [8]	Enqueue, Dequeue, Buffer Overflow, Timer
Network Management	Neighbor/Link/Tunnel Failure Detection, Data-plane State Migration [17], Fast Re-Route [25]	Timer, Link Status
Network Monitoring	Sketches [5, 15], Time Window Functions, Microburst Detection [3], INT [20]	Timer, Enqueue, Dequeue, Buffer Overflow
Traffic Management	AQM [16, 22], Policing, Packet Scheduling [27]	Enqueue, Dequeue, Buffer Overflow/Underflow, Timer
In-Network Computing	Coordination [12], Caching [13]	Timer, Link Status

state is local to a single physical pipeline stage as in the baseline PISA model. Metadata, created by enqueue and dequeue events, propagates through the pipeline (on its own, or alongside arriving packets), allowing processing to proceed at line rate.

While this model is conceptually simple, we need to make sure the pipeline is wide enough to carry all the events, and at the same time, be able to handle all the required stateful operations. For example, suppose we write a P4 program to compute queue sizes. On a single clock cycle, an enqueue event wants to increment the size of queue 0, a dequeue event wants to decrement the size of queue 1, and an ingress packet event wants to read the size of queue 2 in order to make a forwarding decision. Is it possible to support all of these memory operations simultaneously without resorting to multi-ported memory?

Rather than use multi-ported memory we can instead use multiple single-ported register arrays that are suitably coordinated. Packet event read-modify-write operations always operate on the main register that maintains the algorithmic state, the queue size in our example. The read-modify-write operations for enqueue and dequeue events are aggregated in separate register arrays, in the same or potentially a different pipeline stage. During idle clock cycles when there is spare memory bandwidth available, the aggregated operations are applied to the main register that maintains the algorithmic state. Idle clock cycles occur when the workload contains larger than minimum size packets or when the PISA pipeline is configured to run faster than line rate, which is typical in modern switch chips [19]. Figure 3 depicts how this mechanism can be used to process enqueue, dequeue, and packet events to maintain queue sizes.

Stale state. It is important to note that whenever state is distributed across pipeline stages, the algorithmic state will sometimes be stale because of the time it takes state to propagate through a pipeline, or from one pipeline to another. One redeeming feature is that staleness is bounded if the pipeline runs slightly faster than the line rate (as is typical). So, while the state may be temporarily imprecise, the resulting algorithm has well-defined behavior. For example, an application detecting heavy hitters might detect a flow a few nanoseconds late, which is unlikely to matter. On the other hand, some applications require more care (e.g. for consensus algorithms) and the programmer needs to be aware of the staleness. If needed, staleness can be reduced by freeing up processing capacity in the pipeline, for example by not using some of the external ports. This means there is more capacity available to carry metadata from one stage to another, to update algorithmic state. It also opens up another design trade-off: packet processing bandwidth versus accuracy of the data-plane algorithm. This trade-off closely resembles the one provided by sketch algorithms: switch memory versus accuracy of the data-plane algorithm.

When distributing state between stages, we also need to consider how memory accesses are scheduled, depending on which events are the most important and urgent, and whether priorities are assigned by the programmer, the compiler, or the hardware. We plan to address these questions in future work.

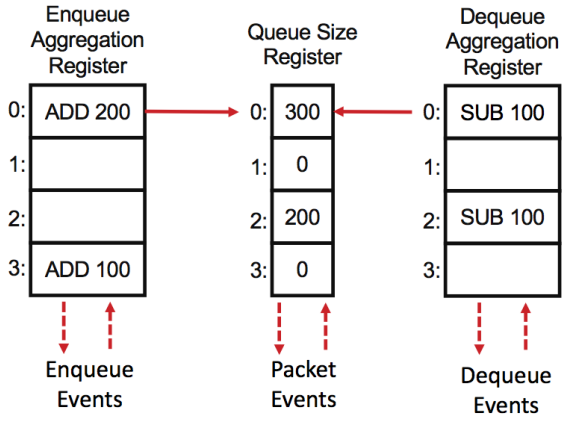

Figure 3: Updating algorithmic state across multiple pipeline stages. The goal: keep the algorithmic state (queue size) up to date. Low-priority enqueue and dequeue events are aggregated in separate register arrays, then applied to the main register when memory bandwidth is available.

5 HARDWARE FEASIBILITY

SUME Event Switch Architecture. To demonstrate that our event-driven architecture is feasible to implement in hardware while processing packets at full line rate, we developed a prototype on the NetFPGA SUME platform using the P4→NetFPGA tools [11]. Our prototype, which we call the *SUME Event Switch* supports regular P4 packet events, plus enqueue, dequeue, and drop events, timer events, link status change events, and a configurable packet generator. Figure 4 shows a block diagram. The Event Merger is responsible for gathering all new events and placing them into metadata that flows through the pipeline. If there are no ingress packets for the metadata to piggyback onto, the Event Merger generates an empty packet, attaches the event metadata and injects it into the P4 pipeline. The pipeline functions themselves are described in P4, and compiled using Xilinx SDNet [28] to run on the FPGA.

The event handling is very efficient, requiring relatively few FPGA resources. Table 3 shows that on a Xilinx Virtex-7 FPGA, the event logic consumes at most 2% additional resources.

In practice. We teach a graduate networking class at Stanford in which students build the hardware and software components of an Internet router; then extend it to add new features of their own choosing. The class uses P4 to define the forwarding behavior. In 2019, the students built their projects on the SUME Event Switch and implemented several different data-plane applications. We highlight a few here.

Liveness Monitoring in the Data Plane. The event-driven programming model was used to implement a protocol in the data plane that periodically checks the liveness of neighboring network devices by transmitting echo request packets and waiting for replies. Upon detecting failure of a neighbor, the data plane transmits notifications to a central monitor, with no intervention by the control plane.

Time-Windowed Network Measurement. A common data-plane task is to compute a function of a signal, such as a moving average, over a sliding window of time. This sort of operation is very natural to implement using timer events. One student group demonstrated how to use timer events in conjunction with a simple shift register to accurately measure flow rates in the data plane.

Table 3: The cost of adding support for events in the SUME Event Switch architecture. The increase in resources are shown as a percentage of the total resources available in a Xilinx Virtex-7 FPGA.

FPGA Resource	% Increase
Lookup Tables	0.5
Flip Flops	0.4
Block RAM	2.0

Computing Congestion Signals. In this project, the students implemented a simple AQM policy to enforce flow-level fairness, similar to FRED [16]. Enqueue and dequeue events were used to compute congestion signals (total buffer occupancy, per-active-flow buffer occupancy, and active flow count). Timer events periodically sample the buffer occupancy and send a report to a monitor which maintains a time series of the buffer occupancy.

Fast Re-Route. Link status change events make it easy to implement fast re-route policies in the data-plane. When a link failure is detected, the prototype updates its forwarding decisions immediately to send packets along a backup route.

Community. We will contribute the SUME Event Switch architecture to the P4→NetFPGA project[2] so that the community can experiment with their own event-driven programs on real hardware.

6 RELATION TO MODERN PISA DEVICES

Today's P4 programmable devices expose a programming model that resembles the one provided by baseline PISA architectures. That is, the only events that are explicitly exposed to the programmer are packet events. Some P4 targets can indirectly support other events as well. For example, Tofino [19] contains a configurable packet generator which the control-plane can configure to generate periodic packets and hence emulate timer events. Tofino also supports packet recirculation, which can emulate dequeue events that trigger the ingress pipeline. However, supporting all of the events listed in Table 1 requires changes to existing hardware.

7 RELATED WORK

Our proposed Event-Driven PISA architecture builds directly upon the baseline PISA architecture described in [2]. The authors of dRMT [4] propose to modify the baseline PISA architecture by disaggregating table memory and compute resources. They demonstrate that this approach leads to higher resource utilization as well as more flexibility when applying match-action tables. However, the programming model that is used to configure their dRMT architecture is identical to the one provided by the baseline PISA architecture. Therefore, our event-driven PISA architecture is able to support the same programs as dRMT.

There has been a number of recent efforts to build new abstractions for programming the network data-plane. Domino [26] introduced the notion of packet transactions which are stateful read-modify-write operations that are performed atomically per packet. This per-packet atomic constraint

[2]GitHub Wiki: https://github.com/NetFPGA/P4-NetFPGA-public/wiki

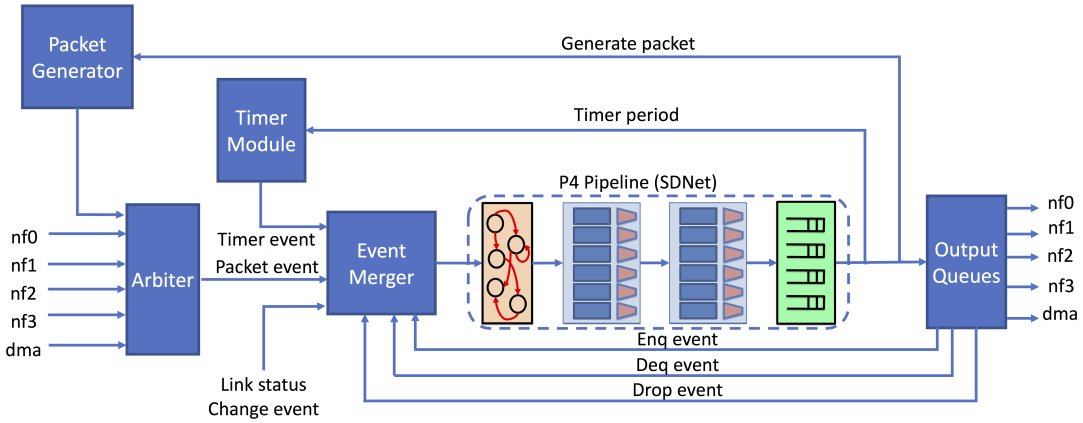

Figure 4: The SUME Event Switch architecture implemented on the NetFPGA SUME platform.

enables Domino to provide consistency guarantees to data-plane programs. However, it also significantly limits the complexity of the read-modify-write operations. As a result, the authors of FlowBlaze [24] propose a new abstraction which distinguishes between global state and flow state. Operations on flow state can be more complex because they only need to complete atomically between packets of the same flow, rather than on a per-packet basis. Both of these proposals only consider single threaded data-plane programs. In an event-driven programming model there can be many event processing threads that share the same state. Defining a consistency model for multi-threaded data-plane programs remains an area of future work.

8 CONCLUSION

All network algorithms are event-driven. As has been shown in this paper, P4 is actually a domain specific language suitable for expressing line rate event processing, not just packet processing. The set of data-plane algorithms that can be expressed in today's data-plane programming model is a strict subset of what can be expressed using our more general event-driven programming model. Events give data-plane programmers much more flexibility, enabling them to implement algorithms that derive and use congestion signals, update state multiple times and independently of packet arrivals and departures, and even compute functions over windows of time much more naturally. However, data-plane algorithms are not the only network algorithms that are event-driven. If one looks at the protocols running in end-host software and in the control plane, it can be seen that they are also event-driven. For example, the state machine for a simple reliable delivery protocol is driven by packet arrivals, packet departures, and timeout events. And the state machines for link state routing protocols are driven by periodic events, timeout events, and link status change events. Since most network algorithms are event-driven, we believe that data-plane architectures should be as well. This approach has the potential to offload much more functionality to high-speed data-plane hardware.

ACKNOWLEDGMENTS

The authors thank Andy Fingerhut, Nate Foster, and Mihai Budiu for fruitful discussions that helped lead to some of the ideas presented in this paper. The authors also thank the anonymous HotNets reviewers whose valuable feedback helped to improve the quality and clarity of this paper. This research is funded by Xilinx Inc, the Stanford Platform Lab, and the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL: sdn EnAbled MeasuRement for alL project (Project Reference EP/P025374/1).

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM CCR*, Vol. 44. ACM, 503–514.

[2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 99–110.

[3] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proc. of the Workshop on Self-Driving Networks*. ACM, 22–28.

[4] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proc. of ACM Sigcomm*. ACM, 1–14.

[5] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[6] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *ACM SIGCOMM CCR* 46, 2 (2016), 18–24.

[7] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking* 4

(1993), 397–413.

[8] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proc. of the Conference of ACM Sigcomm*. ACM, 29–42.

[9] Juha Heinanen and Roch Guérin. 1999. *A single rate three color marker*. Technical Report.

[10] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX NSDI Symposium*. 161–176.

[11] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4-> NetFPGA Workflow for Line-Rate Packet Processing. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 1–9.

[12] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th USENIX NSDI Symposium*. 35–49.

[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[14] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proc. of the Symposium on SDN Research*. ACM, 10.

[15] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: a better NetFlow for data centers. In *13th USENIX NSDI Symposium*. 311–324.

[16] Dong Lin and Robert Morris. 1997. Dynamics of random early detection. In *ACM SIGCOMM CCR*, Vol. 27. ACM, 127–137.

[17] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. 2017. Swing state: Consistent updates for stateful and programmable data planes. In *Proc. of the Symposium on SDN Research*. ACM, 115–121.

[18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of the ACM Sigcomm Conference*. ACM, 15–28.

[19] Barefoot Networks. 2019. Tofino - World's Fastest P4-programmable Ethernet Switch ASICs. (2019). https://barefootnetworks.com/products/brief-tofino/

[20] P4.org. 2018. In-band Network Telemetry (INT) Dataplane Specification. (2018). https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf

[21] P4.org. 2018. P4 Portable Switch Architecture (PSA). (2018). https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[22] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review* 33, 2 (2003), 23–39.

[23] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. 2013. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 148–155.

[24] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. Flowblaze: Stateful packet processing in hardware. In *16th USENIX NSDI Symposium*.

[25] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting emerging applications with low-latency failover in p4. (2018).

[26] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 15–28.

[27] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proc. of ACM SIGCOMM Conference*. ACM, 44–57.

[28] Xilinx. 2018. SDNet. (2018). https://www.xilinx.com/products/design-tools/software-zone/sdnet.html