Challenging the Stateless □ o of Programmable Switches

Nadeen Gebara
n.gebara17@imperial.ac.uk
Imperial College London

Alberto Lerner
alberto.lerner@unifr.ch
University of Fribourg – Switzerland

Mingran Yang
mingrany@mit.edu
MIT CSAIL

Minlan Yu
minlanyu@g.harvard.edu
Harvard University

Paolo Costa
paolo.costa@microsoft.com
Microsoft Research

Manya Ghobadi
ghobadi@csail.mit.edu
MIT CSAIL

ABSTRACT

Programmable switches based on the Protocol Independent Switch Architecture (PISA) have greatly enhanced the flexibility of today's networks by allowing new packet protocols to be deployed without any hardware changes. They have also been instrumental in enabling a new computing paradigm in which parts of an application's logic run *within* the network core (*in-network computing*).

The characteristics and requirements of in-network applications, however, are quite different from those of packet protocols for which programmable switches were originally designed. Packet protocols are typically stateless, while in-network applications require frequent operations on shared state maintained in the switch. This mismatch increases the developing complexity of in-network computing and hampers widespread adoption.

In this paper, we describe the key obstacles to developing in-network applications on PISA and propose rethinking the current switch architecture. Rather than changing the existing architecture, we propose augmenting it with a *Stateful Data Plane* (SDP). The SDP supports the requirements of stateful applications, while the conventional data plane (CDP) performs packet-protocol functions.

CCS CONCEPTS

• **Networks** → **Intermediate nodes**; **Programmable networks**; **In-network processing**; • **Hardware** → **Networking hardware**; **Hardware accelerators**.

KEYWORDS

In-network Computing, Stateful Applications, Programmable Switches, PISA

ACM Reference Format:
Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the Stateless Quo of Programmable Switches. 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, 7 pages. https://doi.org/10.1145/3422604.3425928

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

ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425928

1 INTRODUCTION

The ability of programmable switches based on the Protocol Independent Switch Architecture (PISA) [2, 7, 9, 35] to execute data-plane programs at line rate has opened up new opportunities for researchers and practitioners, spurring unprecedented innovation in network protocols and architectures, e.g., [4, 7, 13, 25, 35, 41, 43].

They have also paved the way for *in-network computing* [11, 38, 48], a new class of applications, ranging from caching [22] and database query processing [26, 47] to machine learning (ML) [33, 40, 52] and consensus [16, 17, 27], that take advantage of the ability to execute arbitrary code within the network core (as opposed to just at the edge), leveraging the switches' unique vantage point. However, while programmable switches have been crucial to enable this new paradigm, we argue that their current architecture is a poor fit for the emerging applications, introducing unnecessary development complexity and impacting performance. This is limiting further growth and precluding widespread adoption of in-network computing, ultimately hurting innovation.

PISA and some of its recent extensions [13, 41] were primarily devised to handle traditional packet-processing operations, e.g., address rewriting or forwarding based on header fields, which typically require limited state. Their architectures, therefore, rely on a pipeline of independent match-action stages across which memory and compute resources are distributed (§2).

In contrast, in-network computing applications tend to be *stateful* and comprise a sequence of operations performed on complex data structures, such as hash tables or caches. Such complex operations are hard to accommodate in current PISA-based switches because the application's state is scattered across pipelines and match-action pipeline stages. For instance, inserting a new item into a cache data structure might require a packet to traverse all match-action stages in a pipeline to check whether that item is already present. If the item is not found, it should be inserted in one of the previous stages but this is not possible because pipelines are feed-forward only. The common workaround is to reinsert (or *recirculate*) the packet into the pipeline, potentially at the cost of program correctness and performance.

We believe the time has come to rethink the hardware platform of programmable switches. We advocate the need for a new architecture designed to support both in-network computing and traditional packet protocols. This is analogous to a similar trend in ML hardware design where general-purpose GPUs paved the way for new architectures designed specifically with ML applications in mind, e.g., Google TPUs [23] and NVIDIA Tensor Cores [34].

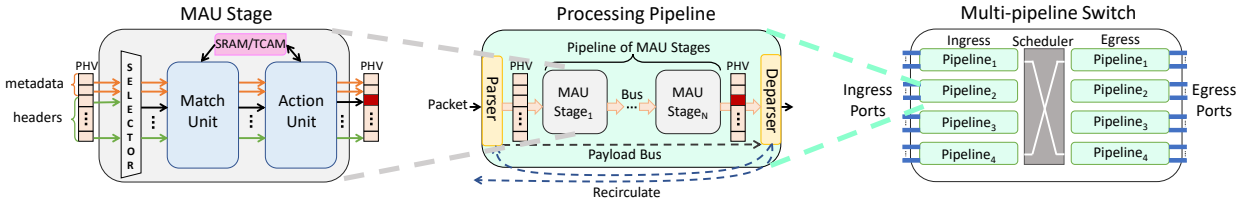

Figure 1: PISA-based switch, adapted from RMT architecture [7]. We show the switch in different levels of detail: Match-Action Unit (MAU) stage within a single switch pipeline (left), switch pipeline of MAU stages (middle), and a switch with multiple pipelines (right).

In this paper, we first provide a comprehensive review of the limitations of existing programmable data planes in supporting stateful applications (§3). Next, we sketch the design of a possible architecture for extending the capabilities of PISA to support stateful applications. We propose to complement the conventional data plane (CDP) in PISA with a new stateful data plane (SDP) (§4). Unlike the CDP, which requires all of its stages to be traversed by packets, the SDP is designed to support stateful applications without causing performance degradation. The key observation driving our design is that today's CDP pipelines have an end-to-end latency of several hundreds of nanoseconds, creating a *slack* time that can be leveraged to perform more complex operations in the SDP while the packet headers traverse the CDP. As long as the stateful operations are completed within the slack time, and no packets are blocked, no performance loss occurs. This could enable supporting stateful operations without disrupting traditional traffic.

While our design is still preliminary and several questions remain to be investigated, we consider this paper a first step towards enabling stateful in-network computing at line rate. We hope to trigger new research and innovation at the boundaries of computer architecture, programming languages, and networking disciplines.

2 PROGRAMMABLE SWITCHES TODAY

Although the micro-architectural details of today's programmable switches vary, they usually comprise four main elements: a parser, a deparser, a pipeline of Match-Action Unit (MAU) stages, and a bus connecting these elements. Fig. 1 shows the components of a generalized programmable switch architecture, commonly referred to as the Protocol Independent Switch Architecture (PISA) [2].

Packet parser, header vectors, and packet deparser. As shown in Fig. 1, the processing pipeline starts with a programmable parser that extracts the packet header fields required for the program execution [18]. These fields are transferred to a large register file called the Packet Header Vector (PHV) [7]. Each MAU stage takes a PHV instance as its input, performs an operation using data in the PHV, and outputs data to the following PHV instance (input of next stage) through the bus inter-connecting the MAU stages. The final PHV instance is connected to a deparser which reassembles the packet and the newly obtained headers.

Match-Action Unit stages. The MAU stages are the processing elements of a programmable switch, as illustrated in Fig. 1. They can "execute" a match-action table; i.e., they select specific fields from the input PHV and compare them to values stored in the match table to determine the actions to be executed. Match tables can be stored in SRAM or TCAM, allowing more sophisticated matching conditions such as longest prefix matching. If a match is found, the action associated with the matching condition is provided to a specific ALU in the form of a simple RISC-like instruction. The action unit has multiple ALUs that operate on PHV fields (header or metadata), and/or register data. The complexity of the operations performed by ALUs varies across switch micro-architectures. For example, MAU stages in Domino [41] support more complex actions (atoms) than those originally proposed in RMT at the expense of increased area. However, the complexity of the operations is limited by the timing constraint imposed within a pipeline stage.

Registers. An action may need to change a value stored in the MAU beyond the lifetime of a packet traversing the switch. For this purpose, PISA switches provide *registers* in MAU stages, usually stored in SRAM. Registers are the only state holders that can be updated through the data plane. However, such operations usually have tight restrictions on the number of registers that can be concurrently accessed within a stage [27].

A pipeline of MAU stages. Multiple MAU stages form a pipeline between the parser and deparser. The MAU stages are independent of each other and *share no resources* as shown in Fig. 1. This design has one important implication: the information is propagated across the stages in a single direction, *feed-forwarding* data through the connecting PHVs. If a computation requires reading/writing values more than once, the packet needs to traverse the pipeline several times. This is referred to as *recirculation*.

Multi-pipeline switches. To scale the number of ports, switch manufacturers use multiple parallel pipelines with no state sharing as depicted in Fig. 1. Ingress (egress) pipelines are statically bound to their respective input (output) ports. Packets can move from their ingress pipeline to any egress pipeline via the switching elements, but moving a packet from one pipeline to another is only possible by recirculating the packet.

3 PISA AND THE BATTLE OF STATE

In this section, we first discuss the shortcomings of PISA in the context of in-network compute (§3.1). We then describe the challenges associated with developing a seemingly simple in-network application (top-k heavy-hitter detection) as a concrete example of the challenges faced by in-network compute programmers (§3.2). Finally, we conclude the section by summarizing the key data structures and building blocks used by in-network compute services;

we discuss how they are currently implemented on PISA-based switches and note the corresponding limitations (§3.3).

3.1 Limitations of PISA

Link rates in data centers have increased by a factor of 100 over the last decade, jumping from 1 Gbps in 2009 to 100 Gbps today. A hypothetical 64×100 Gbps-port would process an aggregate of 9.5 billion packets per second assuming 84-byte packets (including 20 bytes for preamble and inter-packet gap). If such a switch were to retire a packet per clock, it would require an unfeasible 9.5 GHz clock rate. This motivates the need for a pipelined architecture (§2) with a small cycle budget per stage to support high throughput even with today's lower clock rates [54].

A pipelined architecture is well-suited for networking protocols, which are typically characterized by relatively simple operations (e.g., rewrite the packet header or select an output port), but it is a poor fit for stateful in-network applications [38] because of the several implementation constraints that such an architecture introduces as we detail next.

L1 Limited support for complex operations. The first consequence of the limited cycle budget is the difficulty of supporting complex arithmetic operations. Such operations range from "simple" multiplications to more complex ones involving floating-point arithmetic. This impacts such applications as gradient aggregation in ML training [40] and complex queries in database systems [47].

L2 Shared-nothing stage architecture. Stages offer a limited set of operations but compensate by executing several of them in parallel. However, they must be independent: an operation's output cannot be used as the input for another operation in the same stage. This effectively forces every sequential action to be unrolled across several stages. Furthermore, a limited number of registers can be accessed at once within a stage. Stateful register operations might need to be partitioned across stages if this limit is reached [27].

L3 Feed-forward transfer across stages. When the state of a program is partitioned across stages, if an operation requires access to the global state, the packet must travel across all the stages. If, after traversing the pipeline, the packet needs to update a variable in one of the previous stages, the only option is to *recirculate* the packet. However, this operation can affect the application's correctness: by the time the packet is readmitted into the pipeline, the state may have changed, thus possibly invalidating the result of the previous read. Recirculation may also reduce the rate at which new packets access the pipeline, decreasing overall throughput [3].

L4 Fixed memory access pattern semantics. In addition to the challenges described earlier, when the state is partitioned across stages, compilers must impose a strict access pattern on the program's variables. For example, if a programmer defines two variables, a and b, in two different stages, s_1 and s_2, the program is restricted to always accessing them in this order. This forces developers to think in terms of *physical* rather than *logical* memory layout, which reduces programming flexibility.

L5 No state-sharing across pipelines. The above issues are exacerbated when the switch architecture hosts multiple parallel pipelines—a common scenario for modern switches. Given the static binding of switch ports and input pipelines and the lack of inter-pipeline channels, the only way a packet can access an application's state stored in another pipeline is by being recirculated to that pipeline. From an application's perspective, a switch is essentially a multi-core processor, with the caveat that one processor can only communicate with another through recirculation paths.

3.2 Example: Top-k Heavy-Hitter Detection

To illustrate the impact of the limitations described in the previous section on a real application, we consider HashPipe [42], a recently proposed P4 implementation for heavy-hitter detection. We chose this application because, despite the simplicity of the high-level protocol, implementing its required data structure on a programmable switch is particularly complex and requires a compromise between accuracy and performance. This is representative of the challenges faced by developers when they attempt to implement stateful applications on programmable switches.

Program overview. The goal of HashPipe is to identify the set of top-k flows that generate the largest number of packets. This is achieved by using an adaptation of the space-saving algorithm [32]. The algorithm maintains a fixed-size table with k entries, each containing the flow ID (e.g., a hash of the TCP 5-tuple) and the corresponding packet counter. Every time a packet arrives, if its flow ID is already stored in the table, its counter is incremented. Otherwise, the flow ID with the smallest counter is evicted and replaced by the flow ID of the incoming packet, and the previous counter value is incremented by one.

Shared-nothing stage architecture. Determining the flow with the smallest count value requires register accesses and sequential comparison operations that exceed the capacity of a single stage. To circumvent this limitation, the table's registers must be partitioned across pipeline stages. However, since such a solution would restrict the number of table registers (keys) to the number of pipeline stages, HashPipe uses a probabilistic solution. Specifically, it applies d hash functions (one per stage) to the flow ID to inspect d randomly selected entries in different stages, where d is the number of stages. It then determines the minimum value across these selected entries. This approach, however, incurs an accuracy error because the minimum value across just d entries can be far from the minimum value of the entire table.

Feed-forward transfer across stages. Partitioning the table across d stages presents another challenge. A naive way to find the minimum across the d entries is the following. As the incoming packet traverses all stages, the program checks whether its flow ID (f_{new}) is already stored in any of the stages, and it uses the metadata field to keep track of the flow ID with the smallest packet counter encountered thus far (f_{min}). If f_{new} is found, its counter is simply incremented and no further action is required. However, if the packet reaches the last stage and f_{new} has not been found, the packet needs to be recirculated to replace the entry with f_{min}. In the worst-case scenario, the need to recirculate *all* packets can result in halving the overall throughput [42].

To avoid recirculation and the corresponding performance hit, HashPipe modifies the space-saving algorithm by inserting f_{new} in the first stage. The flow evicted in the first stage is carried forward as

Class	Data Structure	Application(s)	Limitation(s)	Implementation trade-off
C1	Bloom filter	Distributed storage systems, network telemetry, load balancing, databases [30]	L2	Limited number of hash functions [22, 33, 41, 47, 53]
	Count-min sketch	Distributed storage systems, network telemetry [53], databases	L2	Limited number of hash functions [22, 41, 47]
	Hash table	Distributed storage system, network telemetry, databases, load balancing, distributed system co-ordination [14, 19–22, 27–29, 33]	L2	Limited collision list & external overflow [26] Lazy eviction threshold tuning required to evict stale hash table entries [12]
C2	Vector	ML training [5, 40, 48]	L1 L2	Limited vector size reduces application goodput Lack of floating point operations (fp) makes accuracy dependent on fp conversion efficiency [40]
	Priority queue	Network telemetry, databases [47]	L2 L3 L4	Depth of queue bounded by pipeline stages requires approximate solutions with reduced accuracy [3, 42]
C3	Cache	Distributed storage systems, network telemetry, databases	L2 L3 L4	Cache policy managed through the control plane & cache update operations violate line rate [22]
	Cuckoo hash table	Distributed storage system, network telemetry, databases,load balancing, distributed system co-ordination [14, 19–22, 26–29, 33]	L2 L3 L4	No known implementation is available
	Matrix	ML inference [43]	L1 L2	No known implementation is available

Table 1: Mapping representative data structures to PISA-based switch architectures. (C1) implementation is practical; (C2) implementation violates line rate or uses approximate solution; (C3) no complete data-plane implementation. Since L5 is not application-dependent, and not a property of the data structure itself, it is not included in the table.

part of the packet metadata field. If a flow ID with a smaller counter is found in the next stage, the two are swapped and the latter is carried forward. Although this approach circumvents the need for recirculation, it can result in duplicate entries if f_{new} is already present in one of the later stages. These duplicate entries reduce the usable table memory and can result in the accidental eviction of heavy flows whose counts are spread across multiple registers. Such evictions cause HashPipe to have a higher false-positive rate than the space-saving algorithm when keys are over-reported [42].

Multiple pipelines. Depending on the definition of *flow*, e.g., the TCP 5-tuple vs. "all packets generated by a tenant," the same flow ID might appear on different pipelines, further complicating a HashPipe-like implementation. One solution could be to force packets to be steered through the same pipeline (ingress or egress) via recirculation. Alternatively, independent tables could be maintained (one per ingress pipeline) but this would accentuate the risk of duplicate keys and, hence, of lower accuracy.

3.3 Stateful data structures in PISA

A top-k list is not the only data structure that is difficult to implement in PISA switches. We surveyed the in-network computing literature and identified the most common data structures used. We classify them in Table 1 according to their implementation difficulty and the impact of the limitations outlined in §3.1.

Bloom filter [6]. A membership test using a Bloom filter with H hash functions requires a *read and combine operation* on the values held in H registers. Such an operation cannot be completed within a single stage because of L2. Therefore, a Bloom filter must span $H + 1$ stages to read the register values and combine their outcome. This bounds the number of hash functions by the number of pipeline stages.

Count-min sketch [15]. A count-min sketch with H hash functions uses the minimum value accessed by a hash function as an estimator of a key. The estimator requires a read and combine operation on H registers; hence, it exhibits constraints similar to those of Bloom filters because of L2. Universal sketches with more complex estimators are implemented by performing sketch updates in the data plane and using the control plane for estimator computation [31].

Hash table. Many hash-table implementations maintain a sequential list to handle collisions. However, because of L2, the list needs to extend across multiple stages. This caps the maximum length of the collision list at the number of stages left after the stage in which the hash computation is performed. A possible workaround is to rely on the control plane to handle overflows but, again, this causes significantly higher latency [26]. An alternative approach is to rely on multiple hash tables with different hash functions used in different stages [12]. Although this is a better solution, it is less memory efficient than using a Cuckoo Hash table [36]. Multi-table hashes such as Cuckoo Hash may seem viable at first. Yet, a pure data plane implementation requires recirculation when an element is evicted from the last table. While the recirculation is ongoing, new elements can be inserted and evicted, creating a possible race condition. Today, implementing Cuckoo Hash tables in PISA needs control-plane support.

Vector and Matrix. Vector operations can be mapped onto a PISA switch in a variety of ways. When the size of a vector exceeds the total number of stateful operations that can be performed on packet headers (L2), applications must partition the vector across multiple packets. For example, since ML tensors are large vectors, gradient aggregation requires applications to send multiple packets to complete a single vector aggregation operation. Further, since L1 does not allow floating-point operations, the provided gradient values must be converted to fixed-point numbers. Careful parameter

tuning is needed to avoid accuracy loss. Moreover, matrix operations required for in-network ML inference [43] are even more complex than vector ones. A typical case may comprise several multiply-accumulate operations [52]. Here, the operation needs to be spread across multiple stages because of L1 and L2.

Cache. Implementing associative caches in the data plane presents challenges similar to the HashPipe example (§3.2). Cache lookups and cache updates of existing keys can be realized by distributing the keys across multiple stages because of L2. However, because of L3, replacing an existing key with a new one can only be done through recirculation, which might lead to correctness violations. An alternative approach explored by recent work [22] is to manage cache insertions and evictions via the control plane. This shows higher memory efficiency than the data-plane implementation and can support variably-sized values but it incurs a performance penalty, as control-plane operations are up to two orders of magnitude slower than data plane operations [49].

4 BRIDGING THE GAP

This section presents our vision for a stateful switch architecture. We first provide an overview of current research efforts and then describe our solution. We do not claim to have all the answers yet. Rather, we hope to start a discussion on new ways to design programmable switches.

PISA variations. There have been attempts to extend PISA switches by improving them along at least three dimensions: *i)* adding more sophisticated atoms (actions), e.g., in Domino [41]; *ii)* allowing more flexible memory access patterns, e.g., in dRMT [13]; and *iii)* mixing MAU stages and "configurable logic-based" stages within a pipeline, e.g., in Flowblaze and Taurus [37, 43]. All these proposals support more expressive programs than a "pure" PISA switch because they relax some—but not all— the limitations discussed in §3.1. While in principle we could combine all these approaches to address the shortcomings of PISA in supporting stateful applications, such an approach would result in longer and more complex pipelines, and would therefore increase the cut-through latency of packets that do not require any stateful operations, and potentially result in resource wastage [54].

Instead of extending the PISA pipeline, we explore an alternative approach that uses *pipeline specialization* and works *alongside a PISA pipeline*. In contrast to other designs that complement the PISA pipeline with external memory resources on servers [24], our design combines the PISA pipeline with an additional tightly integrated pipeline specialized for stateful computations.

Stateful Data Plane (SDP). We introduce a separate data plane that operates in conjunction with the conventional data plane (CDP), as depicted in Fig. 2. The goal of the SDP is to support a more powerful set of stateful operations as first-class citizens.

The two planes are synchronized at the shared parser and deparser units. The packets are parsed normally, as they would be in a PISA switch, but we propose extending the parser graph's semantics to allow the extraction of fields used by the SDP as well. Fields requiring traditional header processing are sent to the CDP through the PHV, whereas fields requiring stateful computations are streamed to the SDP through a separate *Stateful Queue* (SQ).

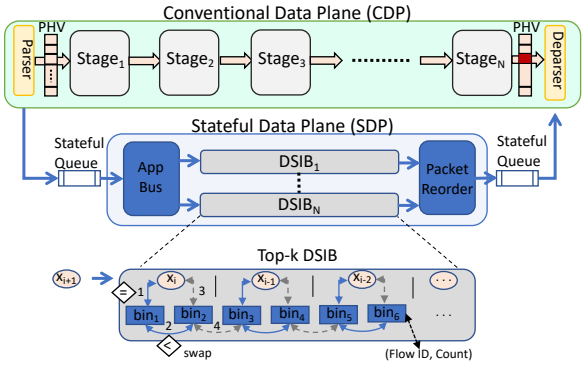

Figure 2: Our vision for the next generation of programmable switches is a Stateful Data Plane (SDP) that computes the state while packets traverse the Conventional Data Plane (CDP). Individual stateful computations can be implemented through specialized blocks (DSIB). We show in the detail how a DSIB calculates the top-k flows using a pipelined hardware algorithm [45].

A series of units called *Data Structure Instance Blocks* (DSIB) are at the heart of the SDP. These blocks house structures that maintain the state of applications and are specialized for performing the required operations. We expect DSIBs to leverage pipelining to sustain high throughput and, as we show shortly, *they can avoid the limitations faced by PISA (§3.1) because they do not need to adhere to the PISA aspects that cause such limitations.* Since some data structures may be common, DSIBs may be shared across applications.

An *Application Bus* directs incoming stateful data to their respective associated DSIBs. The bus is equipped with sufficient buffering to match the rate at which inputs are provided by the parser unit, and ensures that no packets are blocked. The latency of DSIBs might vary depending on the operations they support. Therefore, the *Packet Reorder* unit after the DSIBs ensures that the order with which fields leave both the CDP and SDP is consistent.

The main constraint imposed on the SDP and its DSIBs is that *all operations must be carried out within a slack* time. The slack time is the time budget during which a packet traverses the CDP, and is usually determined at compile time [7]. By imposing this constraint, we ensure that the latency of stateful computations matches that of the original pipeline.

Slack calculation. The key element of our solution is mapping a stateful computation onto the SDP and guaranteeing that its worst-case execution-time (WCET) is *within the slack budget.* There are various methods to calculate the WCET given a program and a platform, including static program analysis [50]. We intend to program the SDP using feed-forward languages such as P4, and extending them to support a richer set of data structures. We suspect that P4 naturally produces *temporally predictable code* due to the structure of the language [39]. Also, commercial P4 compilers are already capable of determining the slack for the PISA processing pipeline at compile time [2]. We expect our compiler extension to be able to do the same for the stateful computations. We note that the same extension could be applied to other languages that share these characteristics, such as NPL [10] or Xilinx PX [8].

Some hardware's characteristics are also essential for timing predictability [46]. Consider PISA, where the MAU stages have static instruction scheduling, lack memory hierarchy, and perform no speculative execution. These mechanisms foster predictability. We have the same design goal for the SDP.

An FPGA-based realization. To explore the design space, we suggest the initial realization of the SDP path through FPGAs. They support pipelined computation implementations and the tight hardware control necessary for timing predictability. Recent FPGAs may have dozens of 100-Gbps hard MAC units and as much as 0.5 GB of on-chip RAM [51]. Also, new techniques are allowing FPGA designs to be clocked from 400 to 850 MHz [44], only slightly slower than today's switching silicons at around 1 GHz. These factors may allow us to achieve an SDP's traversal time in the hundreds of cycles per packet. Such latency is in line with current commercial programmable switches' CDPs, which take at least 800 ns [1].

The question arises as to whether general-purpose CPUs would be a better option with their higher clock rates and easier programmability. The problem with CPUs in our scenario is that they use PCIe lanes to interact with network cards and do not have the bandwidth to support more than a few NICs. Moreover, caching and speculative mechanisms in modern CPUs prevent writing applications with deterministic completion times.

The proposed architecture's exact tradeoffs in terms of chip area, power utilization, and additional costs are still open questions to be addressed. We will research how to minimize the SDPs' overhead while matching the throughput and latency of the conventional pipeline.

An FPGA-based top-k DSIB. Referring back to the top-k example, the parser extracts the packet's flow ID and sends it to the top-k DSIB via the stateful queue and application bus as shown in Fig. 2. This DSIB implements a variation of the space-saving algorithm [32]. It maintains a series of k bins, each storing a flowID (x_i) and a counter, that keep track of the top-k flows [45].

Each bin in the pipeline performs the same operations on a different flow ID entry. We numbered these operations from (1) to (4) in Fig. 2 for convenience. A bin compares x_i's value with the flow ID it holds (1) and updates its counter if they match. Next, the counter value is compared with the next bin's and swapped if the first counter is lower than the second (2). These two operations are then repeated for this next bin (3 and 4), before sending the flow ID entry down the pipeline. Repeating the described sequence of operations as the flow ID traverses the pipeline mimics a bubble sort and sorts the flow IDs in descending order. As a result, when the flow ID gets to the last register without matching any bin value, the last bin's value is automatically updated with the new flow ID and its counter is incremented.

It would not have been possible to access more than one bin in the original CDP due to L1 and L2. Furthermore, even with more powerful atoms that allow two register accesses and a swap operation within a single pipeline stage, the approach is still not possible because register swaps cannot be performed across pipeline stages due to L3.

The original design implementation reports a throughput of 110 million items per second while clocked at a maximum rate of ≈ 115 MHz when implemented on a Virtex-6 FPGA [45]. Considering that the FPGA used is two generations old (40 nm process node), and newer FPGAs typically allow higher clocking rates because of improved technology and advanced optimization heuristics, we anticipate that replicating the design on a state-of-the-art FPGA would result in higher throughput. Furthermore, the proposed design is expected to be capable of maintaining k values in the hundreds of items while meeting a slack value of *at least 800 ns* quoted for programmable switch pipelines [1].

The DSIB block's latency is deterministic given a fixed clock rate and k value (latency = $k \times$ clock_rate/2). We divided by 2 because we can implement two bin operations per clock cycle [45]. By further accounting for the time required to traverse the application bus and re-order unit, we can determine whether the slack time can be met for a specific k-value. Optimizing DSIBs for throughput and slack constraints is a fundamental part of our future research.

Scaling to multi-pipeline switches. We have discussed how to augment a single CDP with a stateful data plane. However, many switches, in particular the high-port-count ones, deploy more than one pipeline. We think that a single SDP path might be sufficient for a 2-pipeline switch, a typical arrangement for today's 32-port switches. However, a 64-port, 4-pipeline switch, may need more SDP paths, potentially one per pipeline. In such scenarios, a *reduction* SDP path may be needed to connect all the SDPs and perform global computations.

5 CONCLUSION

In-network computing represents a new class of applications taking advantage of programmable networks. The first generation of such applications brought benefits to several domains beyond networking, but it also exposed severe limitations in the ability of the current generation of programmable switches to perform stateful computations. In this paper, we identify and characterize the architectural features impacting in-network applications. We curate a selection of data structures commonly used in stateful applications and systematically expose the pain points involved in implementing them.

Armed with insights from this exercise, we suggest a way to extend programmable switches. The cornerstone of our proposal is the deployment of a second data plane in the switch, in addition to the "conventional" one, thus making the switch capable of more complex stateful computations. We propose an initial design for our *Stateful Data Plane* but we hope other researchers join us in this effort by proposing and evaluating alternative designs.

ACKNOWLEDGMENTS

We thank our anonymous HotNets reviewers for their feedback. This work was partly supported by the Microsoft Research PhD scholarship program, by the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/I012036/1, EP/L00058X/1, EP/N031768/1, and EP/K034448/1), by the European Research Council (ERC) under the European Union Horizon 2020 Research and Innovation Program (grant agreements 683253/Graphint and 671653), by NSF grants CNS-2008624 and CNS-1834263, as well as by SystemsThatLearn@CSAIL Ignite Grant.

REFERENCES

[1] Arista. 7170 Series Programmable Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7170-Datasheet.pdf.

[2] Barefoot. Tofino. https://www.barefootnetworks.com/products/brief-tofino-2/.

[3] R. Ben Basat, X. Chen, G. Einziger, and O. Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE 26th International Conference on Network Protocols (ICNP'18)*.

[4] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM '20)*.

[5] G. Bloch. 2019. Accelerating Distributed Deep Learning with In-Network Computing Technology. In *APNET*. https://conferences.sigcomm.org/events/apnet2019/slides/Industrial_1_3.pdf

[6] B. H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[7] P. Bosshart et al. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 99–110.

[8] G. Brebner and W. Jiang. 2014. High-Speed Packet Processing using Reconfigurable Computing. *MICRO* 34, 1 (Jan 2014), 8–18.

[9] Broadcom. BCM56870 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[10] Broadcom. NPL - Network Programming Language. https://nplang.org/.

[11] A. Caulfield, P. Costa, and M. Ghobadi. 2018. Beyond SmartNICs: Towards a Fully Programmable Cloud: Invited Paper. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPRS '18)*.

[12] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford. 2020. Measuring TCP Round-Trip Time in the Data Plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure (SPIN '20)*.

[13] S. Chole et al. 2017. DRMT: Disaggregated Programmable Switching. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*.

[14] E. Cidon, S. Choi, S. Katti, and N. McKeown. 2017. AppSwitch: Application-Layer Load Balancing within a Software Switch. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet'17)*.

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

[16] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soule. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020).

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

[18] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. 2013. Design Principles for Packet Parsers. In *Architectures for Networking and Communications Systems (ANCS '13)*.

[19] R. Harrison, S. L. Feibish, A. Gupta, R. Teixeira, S. Muthukrishnan, and J. Rexford. 2020. Carpe Elephants: Seize the Global Heavy Hitters. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure (SPIN '20)*.

[20] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Proceedings of the Symposium on SDN Research (SOSR '18)*.

[21] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[22] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[23] N. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12.

[24] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM '20)*. New York, NY, USA.

[25] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. 2018. Generic External Memory for Switch Data Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18)*.

[26] A. Lerner, R. Hussein, and P. Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *Proceedings of the Innovative Data Systems Research Conference (CIDR '19)*.

[27] J. Li, E. Michael, and D. R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[28] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*.

[29] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*.

[30] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[31] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*.

[32] A. Metwally, D. Agrawal, and A. El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *International Conference on Database Theory (ICDT '05)*.

[33] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*.

[34] NVIDIA. Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/.

[35] P4.org Architecture Working Group. P416 Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.

[36] R. Pagh and F. F. Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[37] S. Pontarelli et al. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[38] D. R. K. Ports and J. Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*.

[39] P. Puschner and A. Burns. 2002. Writing Temporally Predictable Code. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '02)*.

[40] A. Sapio et al. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR* abs/1903.06701 (2019). arXiv:1903.06701 http://arxiv.org/abs/1903.06701

[41] A. Sivaraman et al. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*.

[42] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*.

[43] T. Swamy, A. Rucker, M. Shahbaz, and K. Olukotun. 2020. Taurus: An Intelligent Data Plane. *CoRR* abs/2002.08987 (2020). arXiv:2002.08987 https://arxiv.org/abs/2002.08987

[44] T. Tan, E. Nurvitadhi, D. Shih, and D. Chiou. 2018. Evaluating The Highly-Pipelined Intel Stratix 10 FPGA Architecture Using Open-Source Benchmarks. In *2018 International Conference on Field-Programmable Technology (FPT '18)*.

[45] J. Teubner, R. Muller, and G. Alonso. 2011. Frequent Item Computation on a Chip. *IEEE Trans. on Knowl. and Data Eng.* 23, 8 (Aug. 2011), 1169–1181.

[46] L. Thiele and R. Wilhelm. 2004. Design for Timing Predictability. *Real-Time Systems* 28, 2–3 (Nov. 2004), 157–177.

[47] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD Conference (SIGMOD '20)*.

[48] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*.

[49] S. Wang, C. Sun, Z. Meng, M. Wang, J. Cao, M. Xu, J. Bi, Q. Huang, M. Moshref, T. Yang, H. Hu, and G. Zhang. 2020. Martini: Bridging the Gap between Network Measurement and Control Using Switching ASICs. In *IEEE 28th International Conference on Network Protocols (ICNP '20)*.

[50] R. Wilhelm et al. 2008. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (May 2008), 36–53.

[51] Xilinx. UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[52] Z. Xiong and N. Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets'19)*.

[53] M. Yu, L. Jose, and R. Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*.

[54] N. Zilberman, G. Bracha, and G. Schzukin. 2019. Stardust: Divide and Conquer in the Data Center Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.