Rethinking the Switch Architecture for Stateful In-network Computing

Alberto Lerner
University of Fribourg
Fribourg, CH

Davide Zoni
Politecnico di Milano
Milan, IT

Paolo Costa
Microsoft Research
Cambridge, UK

Gianni Antichi
Politecnico di Milano
Milan, IT

ABSTRACT

Programmable switches are a disruptive technology that has seen increasing adoption in the past decade. Since their inception, however, there has been tension regarding how to design these switches. Classic programmable switches operate at line rate but impose significant limitations on the expressiveness of their programming models. In contrast, alternative designs relax the strict line rate requirement but are more easily programmable. The common belief is that a switch's performance and its programmability are at odds.

In this paper, we argue that the tension is elsewhere. Many applications use the network to coordinate sets of flows known as *coflows*, while current switches are designed to be individual flow directors. We believe that this conceptual gap—the need to handle coflows rather than independent flows—is what prevents us from creating expressive and fast switch designs at once. We introduce a new device we call an Application-Defined Coflow Processor (ADCP) and discuss how it starts to bridge this gap.

CCS CONCEPTS

• **Networks → Programming interfaces**.

KEYWORDS

Programmable Dataplane, RMT Model

ACM Reference Format:
Alberto Lerner, Davide Zoni, Paolo Costa, and Gianni Antichi. 2024. Rethinking the Switch Architecture for Stateful In-network Computing. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696897

1 MOTIVATION

The Reconfigurable Match-Action Table (RMT) switch architecture has ushered us into a new era of programmable networks [4]. For the first time, it demonstrated that hardware-based switches could be programmed through software and still perform per-packet operations at line rate. RMT switches also demonstrated how limited amounts of data lifted from prior-forwarded packets could be kept on the switch. Using this feature, known as *stateful processing*, a switch program can resort to past contextual information to make better future forwarding decisions. For instance, in a typical data center switch, a traffic-aware load balancing application can maintain flowlet-level information lifted from the packets seen up to that point to make path selection decisions that avoid congestion through load balancing—all in software [20]. Arguably, programmable per-packet operations and stateful processing are the features that allow one to encode traditional networking protocols via software.

These same features have also proven beneficial for applications, allowing cleverly written programs that are not strictly networking protocols to process packets in more general-purpose ways [25]. These programs attempt to manipulate packets based on the their semantics—the reason they are being sent in the first place—rather than strictly forwarding them using their destination addresses. Examples of such application classes that can be offloaded to a switch abound: caching [19], coordination (e.g., locking [33], consensus [7], replication [35]), inter-process communication [22], relational [17, 23, 28] and graph data manipulation [14], and even data aggregation to support machine learning [9, 26, 34]. The list above is far from exhaustive, and more complete surveys can be found elsewhere [21].

Having such a wide range of switch applications may overshadow the challenges developers face in writing them. While many data manipulations are possible in RMT switches, several seemingly trivial ones are currently considered unfeasible [10]. As a result, the community has been striving to create switch designs that can support more expressive *per-flow* oriented programming models. These resulting architecture variations range from fully software-based switches to hardware-based ones with fewer restrictions than RMT.

Application	Coflow Communication Pattern
ML Training and Inference [9, 26, 34]	The weight calculations in a Machine Learning training scenario are distributed across several servers. The servers occasionally engage in an all-to-all exchange of these parameters via aggregation.
Database Analytics [23, 28]	Servers with local storage engage in a pattern of filter-aggregate-reshuffle of data to solve queries over large amounts of data in parallel.
Graph Pattern Mining [14]	Large graphs are partitioned across several servers who then engage in a BSP-style communication exploring increasingly large patterns in the graph at each iteration.
Group Communications [16]	The switch initiates group data transfer within servers running the same application even if some of the servers have different NIC capabilities.

Table 1: Example of applications that benefit from programmable switches and rely on coflows

Some studies document these architectural variations thoroughly (e.g., [2, 12]) but we summarize them briefly as follows. Regarding the software-based category, the most notable example is arguably BMv2 [3]. These switches replace the line rate goal with a *run-to-completion* discipline, which holds a packet in the switch until an arbitrary length computation is completed. Regarding the hardware-based category, Trio [32] is a representative commercially-available example that replaces the notion of processing pipelines with threads. This approach still compromises line rate, even if to a lesser extent than software-based switches. Before Trio, dRMT [5] was another hardware-based variation that added shared memory capabilities on top of an otherwise unaltered RMT switch. That switch, however, was never commercially available.

While it may seem reasonable to improve hardware or sacrifice line speed in order to perform more or more complex computations per flow, we believe that basing new architectures on such attempts is bound to achieve limited success. The reason is that for many relevant applications today, the flows are not independent (§ 2). These applications run on interconnected servers and exchange data through several coordinated flows into what is known as *coflows* [6]. If a switch is to be involved in coflow processing, its architecture needs to accommodate computations that can operate on an entire coflow as an input, manipulate data scattered across its component flows, and produce an output coflow that is different from the input. Various applications that use this coflow paradigm are presented in Table 1.

To make the above architectural assertion more concrete, consider parameter aggregation for machine learning training, arguably one of the most prominent examples of switch programmability. Every server sends the switch a different flow containing a vector of machine learning model weights. The parameter server running on the switch coordinates an aggregation operation among all participating servers over the weights, sending out the results in a very different output flow scheme than the input coflow. RMT switches can implement a parameter server in some form but, as hinted above, it can only do so by drastically restructuring the application to fit the switch [26].

In this paper, we revisit the classic RMT architecture and propose a new design that embraces the notion of coflows by lifting several programmability restriction of the classic RMT model (§ 3). This required addressing several challenges in different areas of the switch.

The first challenge was allowing applications to organize stateful data in the switch arbitrarily. The motivation for this feature comes from the RMT architecture, which forces applications to segregate packets either according to their input port or their assigned egress port. Only by using a method called *recirculation* the flows can be reshuffled arbitrarily but at a great bandwidth and application complexity cost. In contrast, the ADCP offers a new region, called the *global area*, in which applications can easily rearrange coflows arbitrarily without performance loss (§ 3.1).

The second challenge was to break the notion that a packet is a unit of information. In most applications listed in Table 1, a packet holds multiple data elements. For example, in the parameter server case, a packet carries an array of weights, each requiring a separate match-action table (MAT). In some cases, these MATs may each need a table copy, reducing the effective table sizes the switch can hold. The ADCP architecture supports array processing techniques in packet parsing and MATs (§ 3.2).

The third architectural challenge involved breaking another fundamental RMT assumption: that increases in port speeds can be compensated by increasing the clock rate, the minimum packet size, or both. None of these options are sustainable. Clock rates cannot be increased much further, and designing for larger packets penalizes applications, which often transfer little data at a time, e.g., one or a few key/value pairs. We address this issue through a novel pipeline architecture that can handle increasing port speeds without compromising clock rates and/or minimum packet sizes (§ 3.3).

The proposed ADCP architecture may initially appear to require a large amount of area for implementation. However, we have identified several mitigation measures in the design that can make the architecture feasible. Although the complete ADCP design is still in progress, we can already discuss these feasibility measures (§ 4).

The architectural solutions we discuss here are just a subset of what is needed to move past the RMT architectures, and we suggest how the community can take part in addressing them with us (§ 5).

2 BACKGROUND AND RMT LIMITATIONS

We start our discussion by examining specific aspects of the RMT architecture. The reason for choosing RMT is that, out of all the available architecture options, RMT is considered the most established when it comes to processing packets at a line rate. Throughout our discussion, we will focus on the specific regions within the architecture that are affected by the changes we will introduce later in the paper.

The main components of a typical RMT switch are depicted in Figure 1. The servers (not shown) are connected to the switch through n ports, each with a receive (RX) and a transmit (TX) sides (left and right in the figure, respectively). The packets arriving at the RX ports are parsed and buffered independently, and the resulting data from n/p ports is multiplexed into a single *ingress* pipeline (left MUX symbol in the figure). Although we show only two ingress pipelines in the figure (long rectangles on the left), switches with 64 ports tend to have 4 or more.

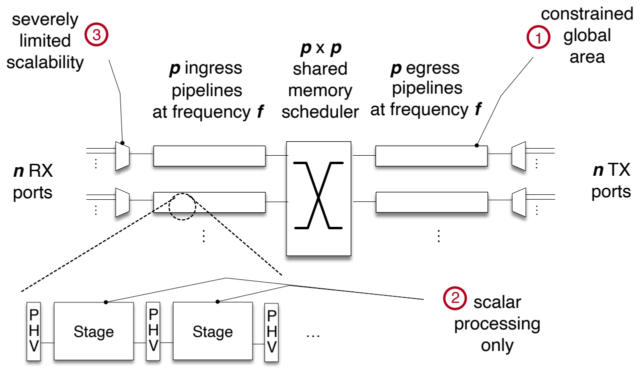
severely limited scalability ③
p ingress pipelines at frequency f
p x p shared memory scheduler
p egress pipelines at frequency f
① constrained global area
n RX ports
n TX ports
② scalar processing only
PHV
Stage
PHV
Stage
PHV
...

Figure 1: RMT architecture and some of its limitations when it comes to processing coflows

In a sense, a pipeline is to a switch what a CPU is to a server. However, while CPUs have shared-memory cores that can communicate freely, pipelines have shared-nothing *stages* arranged in a strict sequence. Each stage communicates with the next through large register files called packet header vectors (PHV) placed between every pair of stages (bottom insert in Figure 1). The PHV naming is misleading; its elements are scalars extracted from the packets.

The pipelines run at a given clock frequency f. This frequency is low compared to CPUs, typically ranging from 1.2 to 1.6 GHz. Given that the switch is line speed, the clock frequency determines the maximum packet rate of a pipeline; a 1.2 GHz one can process 1.2 Bpps. When data arrives at the end of the ingress pipeline, it is deparsed into a packet taking the data modifications into consideration.

The resulting packet is sent to the traffic manager (shown in the middle of Figure 1). The TM is a switching element responsible for forwarding the packet to the pipeline to which its designated TX port is connected. A packet's egress pipeline is calculated on the ingress one. The TM can be implemented as a shared-memory area and work as an output-buffered scheduler [1]. It determines how to forward each packet via a predefined scheduling algorithm and holds them until they can be shipped. Note that forwarding a packet may entail moving it to a different *egress* pipeline than the ingress one from which it came.

The egress pipelines function mostly like their ingress counterparts. The main difference is that, at the end of egress pipelines, the reconstructed packets are demultiplexed across n TX ports (right DEMUX symbol on Figure 1).

As revolutionary as the RMT architecture has been for networking protocols and some applications, there are lacking capabilities that create several issues for other applications, especially those that process coflows (numbered circles in Figure 1). We discuss them in turn next.

① **Coflow Pipeline Semantic.** As briefly noted above, RMT switches lack an area where coflows data could be organized arbitrarily. Figure 2 depicts this difficulty. Regarding the ingress pipelines, coflow data can only be colocated there if the flows come from ports physically attached to the same pipeline. The figure shows why coflow i and j cannot converge on the ingress pipeline.

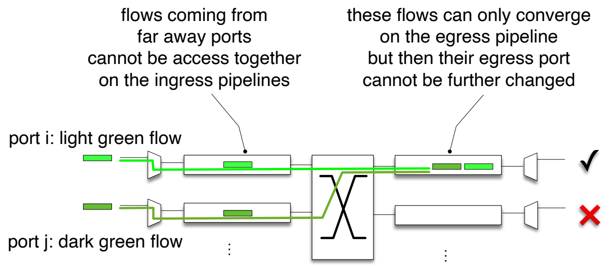
flows coming from far away ports cannot be access together on the ingress pipelines
these flows can only converge on the egress pipeline but then their egress port cannot be further changed
port i: light green flow
port j: dark green flow
✓
✗

Figure 2: Egress-pipeline processing limitations

Flows that need to be processed together could potentially be sent to the same egress pipeline. However, this choice comes with limitations. For instance, the resulting flow can only be output to ports connected to that specific pipeline, as shown in Figure 2. Further, delaying computations until the egress pipeline would forego using the ingress pipeline stages, reducing the total stages involved in the flow's computation by half.

Ultimately, having the results of a coflow operation distributed across RMT egress pipelines predetermines how the resulting coflows can be sent to the servers. A much better option would be sending the results to servers independently of how they are arranged in pipelines.

② **The need for array support.** As also noted above, RMT switches are designed to handle computations over scalar values only. The limitation here is that if a packet carries two or more data elements (e.g., key/value pairs that need to be aggregated), only one element at a time may be used as the input of a match-action table or a register. If we need to match many keys against the same table and those keys came from the same packet, that table must be replicated. Figure 3 portrays this scenario.

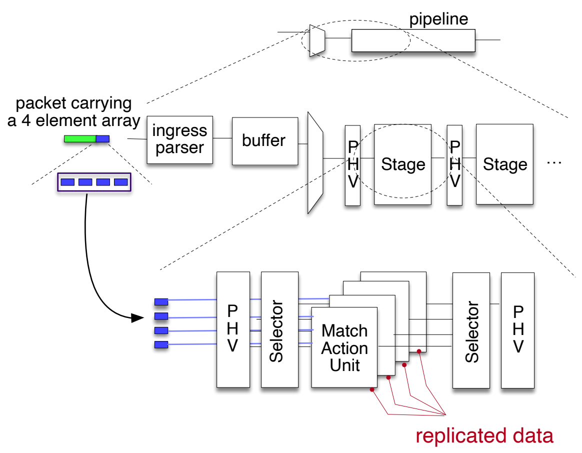

Figure 3: Replication due to scalar processing

Because match-action table memory is scarce and having replicated data would be using it poorly, many RMT applications design their packet formats to carry scalar values only. For instance, the only accurate way to create a hash table over coflows is to use scalar-data packets, to the best of our knowledge. These single-input packets are often small and thus have subpar *goodput*. They also severely limit the throughput of the operations because, even though current RMT-based switches have 12.8 Tbps throughput, they can "only" process 5-6 billion packets per second. The switches, however, do have 16 match action units per stage. In other words, requiring an application to go scalar misses a potential 16× performance boost.

③ **Scalability.** A third issue with RMT switches we need to address is loosely related to coflows. The problem is that increasing port speed in RMT setups can be challenging. To understand why, let us analyze how increasing the switch throughput constraints its pipelines' frequency. The original RMT paper proposed a single pipeline of 64x 10 Gbps ports. This is viable because the combined traffic amounts to a maximum of around 952 Mpps. Therefore, running this pipeline at 952 MHz can achieve line speed. As the ports speed increases, fewer ports can be multiplexed into a single pipeline, lest its frequencies be kept in check. For instance, 64x 100 Gbps ports can generate just about 9.5 Bpps. Clearly, a 10 GHz processor is not a viable options in this scenario.

To support faster port speeds, newer switches resort to a combination of (a) multiplexing fewer ports per pipeline, (b) using more pipelines, and (c) increasing the assumed average packet size, which caps the maximum packet rate. Table 2 illustrates some common compromises to maintain clock speeds at 1.25 or 1.62 GHz, to pick some arbitrary but reasonable values.

Switch Throughput	port speed (Gbps)	# of pipelines	ports per pipeline	minimum packet (B)	pipeline freq. (GHz)
640 Gbps	10	1	64	84	0.95
6.4 Tbps	100	4	16	160	1.25
12.8 Tbps	400	4	8	247	1.62
25.6 Tbps	800	8	8	495	1.62
51.2 Tbps	1600	8	4	495	1.62

Table 2: Port multiplexing poor scalability

The table shows that for 100 and 400 Gbps port speeds, switches needed to increase the minimum packet size and reduce the number of ports per pipeline but remained viable. To reach the next level of bandwidth and port speed, the minimum packet may raise to 495 B, and, even so, only four 1.6 Tbps ports would fit into a 1.62 GHz pipeline. This path is not sustainable.

The reason we bring up issue ③ is that it seems impractical to focus on coflow needs in our ADCP architecture without solving the port speed scalability problem. Fortunately, the design modifications made by ACDP for coflows have also made it feasible to tackle the scalability issue.

3 ADCP HARDWARE ARCHITECTURE

The ADCP architecture retains many aspects from RMT, mainly the ones connected to preserving line rate, but brings a small number of very fundamental changes that address the concerns and issues raised above. Figure 4 depicts our proposed architecture. In the figure, we have highlighted the proposed changes in red to make it easier to identify them.

The first notable change is the introduction of a second traffic manager. This change essentially creates a central pipeline on the switch and gives it properties missing from both the ingress and egress pipelines. In this central pipeline, packet data from different flows can be manipulated together without any of the constraints we discussed previously, addressing the issue ①. We call this region of the switch the *global partitioned area* (§ 3.1).

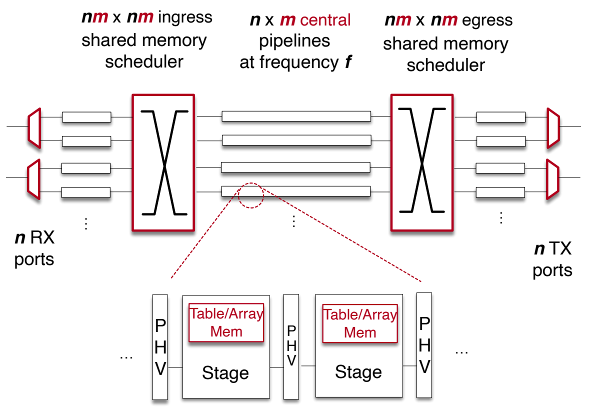
nm x *nm* ingress shared memory scheduler
n x *m* central pipelines at frequency *f*
nm x *nm* egress shared memory scheduler
n RX ports
n TX ports
PHV
Table/Array Mem
Stage
PHV
Table/Array Mem
Stage
PHV

Figure 4: The proposed architecture with modified or augmented areas in red

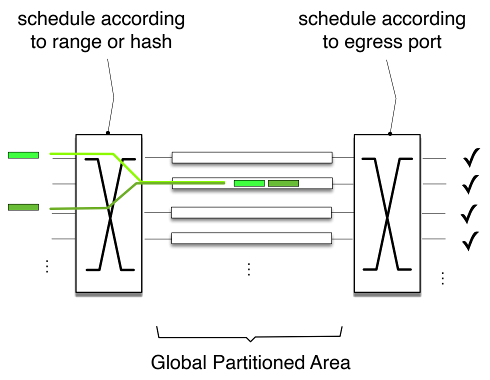
schedule according to range or hash
schedule according to egress port
Global Partitioned Area

Figure 5: Independent processing and forwarding thanks to the global partitioned area

The second modification was made at the pipeline level in response to ②. We have added a special memory area in each stage that can be accessed by all the match action units simultaneously. This memory area allows treating the previously independent match-action units as a unit capable of matching an array at once (§ 3.2).

The third modification revolves around demultiplexing ports into two (or more) pipelines rather than, as in RMT, multiplexing them. Note that the muxes in Figure 1 appear as demuxes in Figure 4. This change addresses the scalability issue ③ and presents unique scalability opportunities (§ 3.3).

3.1 Global Partitioned State Support

The global partitioned area is created by adding a second traffic manager, forming a central set of pipelines. It is deemed *global* because an application can place data across its pipelines without compromising future forwarding options. As Figure 5 shows, the first traffic manager can be used to reshuffle data, for instance, by ranges or hashes over a given data element on each packet. The second traffic manager can then forward the data to any egress port, which, as seen in Figure 2, is impossible with an RMT pipeline scheme.

There is at least one interesting observation and one opportunity about this design. The pipelines used here are, just as in RMT, independent of each other, which is why we consider the area *partitioned*. Therefore, the application needs to define the criteria by which the first TM will forward packets across the pipelines. The observation is that this criteria is most likely different from the one used by the second TM.

While the second TM is more likely to behave as a classic scheduler, the first TM could have better application capability. We may want to use the first TM to, for example, impose an order to packets based on application criteria. This is not to say that the first TM can do general-purpose sorting, but it could keep a sort order while it merges flows that are themselves sorted. The opportunity is that we can expand the semantics of what we consider scheduling in the TM.

Using our parameter aggregation application example, this means that we can place a given weight to aggregate on a pipeline based on the weight's ID hash. However, this choice does not force us to output the aggregated weight to the port connected to that pipeline. Thanks to the second traffic manager, we can forward the aggregated weight to any port, or even to multiple ports.

3.2 Array Support

A global area is useful but does not solve the issue that packets may carry data arrays, with each array element needing to be matched against the same match-action table. Therefore, the ADCP architecture allows a group of match-action units within a stage to simultaneously match an array of values. It does so by interconnecting the match table memory of several match action units. Figure 6 depicts such an arrangement.

The main challenge with this design is to avoid adding physical memory to the stages. Instead, we aim to enable each match-action unit to have its local table memory in non-array scenarios, while also interconnecting the local memories to be configured to handle parallel lookups in array-matching scenarios. This might involve incorporating a programmable interconnect across the table memories that could switch between local and array access patterns.

Up until now, we have focused our discussion on functionality improvements, but this array-matching capability can have tremendous benefits for performance, goodput, and space efficiency inside the ADCP device. For applications, the performance of a switch is connected to the rate of *keys* rather than the packets it can process. RMT Switches with 12.8 Tbps can handle between 5 to 6 Bpps because of the

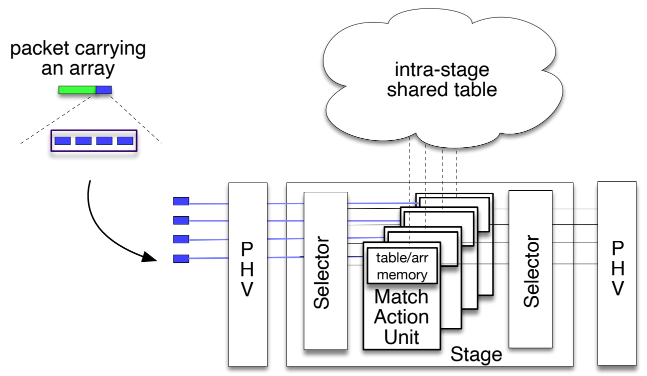

Figure 6: Support for array operations via intra-stage shared memory

compromises made in the name of pipeline clock frequency (cf. Table 2), and RMT switches force a 1:1 relationship between keys and packets. Therefore, any application logic we perform on that switch will be capped at 6 Bops/s. By supporting 8- or 16-wide array processing, the ADCP architecture can push that limit by one order of magnitude simply by allowing the application to pack 8 or 16 keys per packet.

3.3 Faster Ports Support

In the previous sub-sections, we discussed architectural features motivated by coflow support. These changes, however, do not address the scalability issue we previously raised (cf. Table 2). We need to equip the ADCP platform with the capability to evolve along with port speeds.

The strategy used by ADCP to address scalability is deceptively simple. Unlike RMT switches, ADCP divides each port into m pipelines instead of the other way around. This means that the traffic in these pipelines runs at $1/m^{th}$ of the port speed, allowing them to operate at a significantly lower frequency. At the end of the egress pipeline, the pipelines are multiplexed back into high-speed flows. Table 3 shows the benefits of demultiplexing 800 Gbps and 1.6 Tbps ports by 1:2.

port speed (Gbps)	ports per pipeline	minimum packet (B)	pipeline freq. (GHz)
800	8	495	1.62
800	0.5	84	0.60
1600	4	495	1.62
1600	0.5	84	1.19

Table 3: Port demultiplexing examples

Ultimately, the port demultiplexing approach had the potential to future prove this architecture by solving issue ③. For instance, consider the upcoming 1.6 Tbps bandwidth ports. Each of these ports can deliver around 2.38 Bpps using the smallest Ethernet packet. If we want to retire one packet per cycle at line speed, we require each pipeline to work at 2.38 GHz. Alternatively, a design may assume that packets are much larger, as the table shows. This allows having more ports per pipeline (e.g., 8), but these gains do not hold at higher speeds when lower multiplexing factors have to be used (e.g., 4) to keep clock rates at a reasonable level. By demultiplexing a port at a 1:2 ratio, we can reduce the clock speed by half, which can mean the difference between an unfeasible and a practical chip design (§ 4).

Note that port demultiplexing is not without implications. For instance, parsing still needs to be done at port speed, but parsing efficiency is linked to the complexity of structure within packets rather than port speed [11]. Second, as part of parsing, an application must define how to separate the packet contents into m pipelines. For another instance, demultiplexing ports puts pressure on the traffic manager to handle a much larger number of pipelines. We anticipate that this number will increase to 64 in 51.2 Tbps switches and double for 102.4 Tbps, but this will keep clock rates in the same range as today's.

4 FEASIBILITY DISCUSSION

Supporting the features described in the previous section require packing additional logic in the switch chip. While we have not completed a full chip design yet, in this section, we outline some important aspects of such design that will work in our favor and those that will make it more challenging.

The good news is that a significant portion of the ADCP architectural elements can run on a clock frequency that is a fraction of what RMT chips use today. This is important because most line rate architectures tie the base clock frequency to the maximum packet rate supported (cf. Table 2). As we discussed in that table, this trend is not sustainable anymore. In the ADCP, the areas that can benefit from a lower clock include at least the ingress, central, and egress pipelines, thanks to the demultiplexing. Once again, translating the lower frequency into specific benefits requires a more thorough design, but speculatively, it can lower the power requirements of the resulting chip. Lower frequency can also translate into using potentially smaller gates and, therefore, improving the area requirements. Thus, from a frequency standpoint, we do not consider this a challenge to current design practices and fabrication processes.

The first challenge that we instead expect in our design is that we are connecting many more elements and using a wider interconnect. The bigger example is the connection between the traffic managers and the adjacent pipelines. Our main concern here is routing congestion. In modern digital integrated circuits, the routing congestion problem occurs when a large number of wires are routed in a narrow physical space and its resolution can significantly delay the sign-off of the circuit. In general, the routing congestion problem mainly affects the signal wires because they are routed after the power delivery and the clock tree networks, and therefore, they are subject to additional routing constraints.

To ease the routing, modern electronic design automation (EDA) tools organize the floorplan in a grid of so-called *g-cells* and iteratively solve the routing problem using congestion-driven heuristics where the routing congestion is measured as the area of each g-cell divided by the area required to route all the signal wires willing to traverse the cell. Notably, the routing congestion problem is most likely to occur in the proximity of heavily shared intellectual property (IP) blocks, e.g., shared memories, due to the high number of wires that are expected to be routed to the input-output interface of the specific IP.

Back to the ADCP design, the traffic managers represent a possible source of routing congestion. They offer a shared memory area between the central and the ingress/egress pipelines. To minimize the congestion, it is important to avoid monolithic and area-efficient designs for that component. Instead, their floorplan should be spread across the layout and interleaved with other logic elements, e.g., pipelines.

Another expected source of design challenges involves the array processing capabilities as we need to support several match-action tables (MAT) within a stage to perform parallel lookups against a unified MAT memory (§ 3.2). We have several design options in this case. For instance, we can leverage the lower clock frequency of the pipelines and clock the MAT table memory at a much higher frequency. If we wish to support an array width of n, that memory could be clocked n times faster than the pipeline. The lookups in this solution would be done one at a time, but thanks to the clocking difference, we could retire n lookups at once from the point of view of the pipeline.

Naturally, this multi-clock design increases the complexity of the architecture. Furthermore, this design links the memory frequency with the array width we aim to support, which could potentially restrict scalability in future versions of the architecture. Before reaching a decision, we are assessing different area-performance implementations based on various representative application scenarios.

5 A CALL TO ARMS

The research on programmable switches is at an interesting inflection point. A few years ago, the RMT model went commercial and gained such attention that a major chip manufacturer acquired the startup behind that effort. Since then, despite the intrinsic challenges and limitations to support a broad class of in-network compute applications [10], the research community has been tacitly resistant to investigating other design alternatives under the assumption that a valid option was off-the-shelf and new options were unlikely to be manufactured. However, in an interesting turn of events, the commercial RMT switches were recently "retired and discontinued" [15]. The field for replacement technologies is open, and in the past months, we have already seen a few proposals from the industry [29, 32] and the research community [8, 30].

However, all these solutions are still fundamentally offering the same packet-based abstraction provided by RMT and networking devices in general. In contrast, we argue that the recent sequence of events presents a unique opportunity to go back to the drawing board and re-think the switch architecture from the grounds up, explicitly targeting stateful in-network computation [13, 18, 19, 24, 25, 31] as a first-class citizen (along with traditional networking operations). In particular, we should expand the switching capabilities from simple packet processors to *coflow processors* that understand application processing patterns and can bridge the semantic gap between the applications and the networking world.

We hope the design options explored here resonate with other colleagues and will spur fertile discussions and innovations across the research community and industry. There is much to be discussed about the architectural features we presented and beyond. For instance, we believe intriguing opportunities can be unleashed when making the scheduler programmable [27], especially in an architecture like the one proposed here that heavily relies on multiple shared memory schedulers. Further, supporting array processing would require appropriate extensions to the programming model: understanding how such a new hardware primitive would impact programmability is an open question we wish to discuss with the broader community.

ACKNOWLEDGEMENTS

This work has received funding from the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (grant agreement No. 101092908). This work was also partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP D43C22003080001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

REFERENCES

[1] Mutlu Arpaci and John A. Copeland. 2000. Buffer management for shared-memory ATM switches. *IEEE Communications Surveys & Tutorials* 3, 1 (2000), 2–10. https://doi.org/10.1109/COMST.2000.5340716

[2] Roberto Bifulco and Gábor Rétvári. 2018. A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. https://doi.org/10.1109/HPSR.2018.8850761

[3] bmv2 [n. d.]. P4 Behavioral Model (BMv2). https://github.com/p4lang/behavioral-model.

[4] 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. *SIGCOMM Comput. Commun. Rev.* 43, 4 (2013), 99–110. https://doi.org/10.1145/2534169.2486011

[5] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. https://doi.org/10.1145/3098822.3098823

[6] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. https://doi.org/10.1145/2390231.2390237

[7] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. https://doi.org/10.1145/2774993.2774999

[8] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. 2022. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 635–649. https://www.usenix.org/conference/nsdi22/presentation/feng

[9] Nadeen Gebara, Paolo Costa, and Manya Ghobadi. 2021. PANAMA: In-network Aggregation for Shared Machine Learning Clusters. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

[10] 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)*. https://doi.org/10.1145/3422604.3425928

[11] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*. https://doi.org/10.1109/ANCS.2013.6665172

[12] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2023. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications* 212 (2023). https://doi.org/10.1016/j.jnca.2022.103561

[13] Jingqi Huang, Jiayi Meng, Iftekharul Alam, Christian Maciocco, Y. Charlie Hu, and Muhammad Shahbaz. 2022. Accelerating 5G (Mobile Core) Control Plane using P4. P4 Workshop – https://opennetworking.org/wp-content/uploads/2022/05/Jingqi-Huang-and-Jiayi-Meng-Final-Slide-Deck.pdf.

[14] Rana Hussein, Alberto Lerner, Andre Ryser, Lucas David Bürgi, Albert Blarer, and Philippe Cudre-Mauroux. 2023. GraphINC: Graph Pattern Mining at Network Speed. *Proc. ACM Manag. Data* 1, 2 (2023). https://doi.org/10.1145/3589329

[15] Intel. [n. d.]. Discontinuation Notice of Tofino 2 chips. https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/ordering.html.

[16] Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. 2024. Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs. *Proc. ACM Manag. Data* 2, 1 (2024). https://doi.org/10.1145/3639291

[17] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. 2022. P4DB - The Case for In-Network OLTP. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. https://doi.org/10.1145/3514221.3517825

[18] 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 Symposium on Networked Systems Design and Implementation (NSDI 18)*. https://www.usenix.org/conference/nsdi18/presentation/jin

[19] 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 (SOSP '17)*. https://doi.org/10.1145/3132747.3132764

[20] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. https://doi.org/10.1145/2890955.2890968

[21] Somayeh Kianpisheh and Tarik Taleb. 2023. A Survey on In-Network Computing: Programmable Data Plane and Technology Specific Applications. *IEEE Communications Surveys & Tutorials* 25, 1 (2023), 701–761. https://doi.org/10.1109/COMST.2022.3213237

[22] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference ((USENIX ATC 19))*. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

[23] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*. https://www.cidrdb.org/cidr2019/papers/p142-lerner-cidr19.pdf

[24] Heng Pan, Penglai Cui, Ru Jia, Penghao Zhang, Leilei Zhang, Ye Yang, Jiahao Wu, Jianbo Dong, Zheng Cao, Qiang Li, et al. 2022. Libra: In-network Gradient Aggregation for Speeding up Distributed Sparse Deep Training. *arXiv e-prints* (2022), arXiv–2205. https://arxiv.org/pdf/2205.05243

[25] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. https://doi.org/10.1145/3317550.3321439

[26] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation ((NSDI'21))*. https://www.usenix.org/conference/nsdi21/presentation/sapio

[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 *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 44–57. https://doi.org/10.1145/2934872.2934899

[28] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. https://doi.org/10.1145/3318464.3389698

[29] Trident4 [n. d.]. Broadcom Trident4 Chipset. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[30] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 651–665. https://www.usenix.org/conference/nsdi22/presentation/xing

[31] Zhaoqi Xiong and Noa 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)*. https://doi.org/10.1145/3365609.3365864

[32] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. https://doi.org/10.1145/3544216.3544262

[33] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. https://doi.org/10.1145/3387514.3405857

[34] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *SIGCOMM Comput. Commun. Rev.* 54, 1 (2024), 2–21. https://doi.org/10.1145/3687230.3687232

[35] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.* 13, 3 (2019). https://doi.org/10.14778/3368289.3368301