Relaxing state-access constraints in stateful programmable data planes

Carmelo Cascone*
Open Networking Foundation
carmelo@opennetworking.org

Roberto Bifulco
NEC Laboratories Europe
roberto.bifulco@neclab.eu

Salvatore Pontarelli
CNIT/Univ. Roma Tor Vergata
salvatore.pontarelli@uniroma2.it

Antonio Capone
Politecnico di Milano
antonio.capone@polimi.it

ABSTRACT

Supporting programmable stateful packet forwarding functions in hardware requires a tight balance between functionality and performance. Current state-of-the-art solutions are based on a very conservative model that assumes worst-case workloads. This finally limits the programmability of the system, even if actual deployment conditions may be very different from the worst-case scenario.

We use trace-based simulations to highlight the benefits of accounting for specific workload characteristics. Furthermore, we show that relatively simple additions to a switching chip design can take advantage of such characteristics. In particular, we argue that introducing stalls in the switching chip pipeline enables stateful functions to be executed in a larger but bounded time without harming the overall forwarding performance. Our results show that, in some cases, the stateful processing of a packet could use 30x the time budget provided by state of the art solutions.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; • **Hardware** → **Networking hardware**;

KEYWORDS

Network data plane, programmable switch, SDN, NFV

1 INTRODUCTION

Processing large network traffic loads requires the execution of complex algorithms in the network data plane, both to implement smart traffic forwarding policies at line rate or to offload processing that used to happen on general purpose CPUs. To meet the required performance, while still providing the ability to quickly adapt the algorithms to new emerging needs, a new generation of programmable switching devices has been proposed [14, 17].

The challenge of these devices is to provide both programmability and high-performance forwarding. RMT [14] is an example of a hardware design that can achieve high throughput while providing a programmable stateless Match-Action Table (MAT) abstraction, similar to the one provided by OpenFlow [18], but with programmer-defined protocol fields and forwarding actions. Sivaraman et al. [24] demonstrated that it is also possible to implement a high performance *programmable stateful data plane*, provided that strict constraints on the per-packet processing time are met. Here, the challenge is augmented by the need to guarantee state consistency.

*Work done when at Politecnico di Milano

More specifically, a high performance hardware data plane typically processes packets in parallel, using a pipelined design. Each pipeline's stage performs a few operations on a packet, and all stages are executed simultaneously. When the hardware's clock ticks, each packet in the pipeline is moved to the next stage, the packet in the last stage exits the pipeline and a new packet enters in the first stage. In a stateful data plane, a stage performs operations that can read and write state. If more stages can access the same portion of the state a *data hazard* arises. In particular, the longer is the time between a state read and a state write, the higher is the risk of having a state inconsistency. That is, the state read by a stage is going to be invalidated by a write performed in a later stage.

In [24], performance and consistency are guaranteed by ensuring that state read and write happen within one stage, with no state shared between different stages. The cost of this design is the inability to express operations that take longer to complete. In fact, it may not be possible to perform such operations within the limited time budget of a single pipeline's stage. For reference, and assuming that each stage is executed in a single clock cycle, the time budget for an operation would be 1 ns for a pipeline clocked at 1 GHz.

While it would be possible to partition over multiple stages operations that take longer to complete, the consistency harm associated to accessing the same state from multiple stages prevents the application of such techniques. This limitation finally hinders the ability to implement some applications or, in the best case, forces programmers to run approximated version of their algorithms [23].

The mentioned pipelines are usually designed for a worst case scenario, making two stringent assumptions. First, they assume packets are all minimum size. However, a data plane pipeline performs algorithms only acting on packets' header. For a given line rate, larger packet sizes actually mean a lower rate of packet headers to process. Hence, more time per packet can be used to execute the pipeline operations. Second, they assume that the processing of different packets requires access to the same portion of the state. In practice, applications usually access different portions of the state when processing packets belonging to different flows [20].

Starting from these observations, in this paper we try to understand to which extent such assumptions can be relaxed in real-world scenarios. Specifically, we make three contributions. First, we build and make available a trace-based simulator of a variable length pipeline that allows different stages to access the same state [9]. Second, we evaluate the probability of incurring in a state inconsistency, for different pipeline lengths, when processing 6 real traffic traces from both carrier and datacenter networks, taking into account

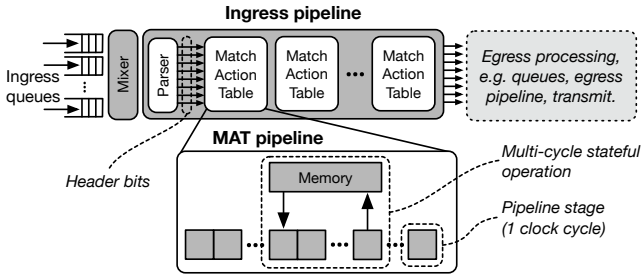

Figure 1: Simplified data plane architecture.

packet sizes and different flows granularity. Our results show that for several traces, and for different flows granularity, the probability of longer multi-stage pipelines to experience an inconsistent state access is relatively small. Therefore, as third contribution we demonstrate that a simple locking scheme can prevent state inconsistencies, while still providing line rate performance in many cases, for the tested traffic traces.

Our work highlights that the maximum length of the pipeline is strictly dependent on the particular traffic trace characteristics and on the considered flow granularity. This suggests that a one-size-fits-all solution is generally inefficient in this area, and that designs that take into account the properties of the workload can provide great benefits. Implementing a full pipeline that takes advantage of the provided observations remains a challenging task we did not address yet. However, we provide synthesis results for one of the required building blocks, i.e., the locking scheme, showing that in principle it can be implemented with little resources overhead. We believe our contributions raise interesting discussion points, in a field that is increasingly looking into reconfigurable hardware to meet application requirements.

2 REFERENCE MODEL

This section presents the data plane architecture we use as reference model and that is implemented by state-of-the-art solutions such as RMT [14]and Cavium's XPliant Packet Architecture [10].

In such architectures (cf. Fig. 1), packets received from the input ports are enqueued in per-port queues and served, with a round robin policy, by a mixer that feeds them to the ingress pipeline. The latter is composed by a programmable packet parser [14] and by a variable number of Match-Action Tables (MATs). For each new packet, the parser extracts the *headers* that are then processed by the MATs elements. A MAT element is itself a pipeline, whose structure may change from one implementation to the other [21, 24],

After the ingress pipeline, the packet is handled by the egress processing stages, such as output queues, scheduler, and an egress pipeline (similar to the ingress one), before being delivered to the output ports. In this paper, we focus on the MAT element's pipeline, since it is the component that executes stateful operations.

We can ignore most of the details of the actual MAT element and model its internal pipeline using just a sequence of stages plus state. Each stage performs a limited amount of operations, such as state read/write, or some computation. Furthermore, to simplify exposition, we assume each stage executes in 1 clock cycle. For stateful operations, the number of stages between a state read and a state write determines the time necessary to process a value read from state, before writing it back. Intuitively, the larger is the number of pipeline stages, the higher the chances of receiving a new packet whose processing requires access to that same portion of the state.

Finally, for the pipeline to be able to process one packet each clock cycle, similarly to previous work [14, 24], we assume that state associated to stateful functions can provide a throughput of one read/write operation per clock cycle.

3 RETHINKING DESIGN ASSUMPTIONS

Current stateful data plane architectures, such as Banzai [24], eliminate data hazards by executing memory read and write operations within the same stage, i.e., in one clock cycle. Such a design derives from the worst case assumption that all packets have minimum size, that they arrive back-to-back, i.e., with no inter-packet gaps, and that they all need to access the same memory area. More specifically, let's consider a data plane with a throughput of 640 Gb/s, with a chip clocked at 1 GHz, as it is the case of RMT [14]. We can assume that packets are read in chunk of at most 80B (i.e., 80×8 bit $\times$ 1 GHz = 640 Gb/s) when entering the pipeline. Consequently, it will take 1 clock cycle to read packets with minimum size $\leq$ 80B, while it will take more cycles to read longer packets, e.g., 19 cycles for 1500B.

The data plane's pipeline is dimensioned to accept a new packet's headers at each clock cycle. However, even when all packets arrive back-to-back, the variability of the packet size will cause the pipeline to experience one or more idle cycles. For example, in the case of 1500B packets, the pipeline will receive packets' headers at intervals of 19 clock cycles.

We observe that packets produced by today's applications have very variable size distributions. E.g., spanning from 64B to 1500B, in a typical case, which could leave some space for relaxing the constraint on the memory read and write operations, when dealing with non corner-case traffic loads.

Flow-level parallelism A second observation is that data plane state can be divided in two types: *global state* and *flow state*. The first type is accessed by all packets, with no distinction, while flow state is only accessed by packets belonging to the same flow, where the definition of a flow is application-dependent. For instance, it could be the L4 5-tuple, the IP source-destination addresses pair, the destination IP address only, a portion of the latter, e.g., a /16 subnet, or even a switch-internal metadata, such as the packet's ingress or egress port. As a matter of fact, the notion of flow state is at the base of existing abstractions such as OpenState [13] and FAST [19], which extend the processing of a MAT with stateful capabilities. Intuitively, if we organize the state as an array, where each cell contains the state of a specific flow, it follows that multiple packets accessing different cells can be processed in parallel, with no harm for consistency. That is, for packets accessing only flow state it is safe to concurrently access the memory.

In practice, functions can use either a combination of the two state types, or just one. For example, a stateful firewall maintains state for each bi-directional L4 flow. A dynamic source NAT maintains both flow state, for each L4 connection, and global state, to store a pool of available ports. Advanced load balancing schemes, such as CONGA [11], maintain flow state at different aggregation levels: i) 5-tuple, to distinguish between flowlets (burst of packets) of the same L4 connection; ii) tunnel ID, to maintain real-time utilization levels of several paths, and iii) global state, to store the best path among the available ones.

Trace	Source	Pkts	Num flows per 1m pkts		
			5-tuple	*ipdst*	*ipdst/16*
CHI15	CAIDA [1]	3.5b	100.6k	57.7k	4.6k
SJ12	CAIDA [2]	3.6b	429k	17k	2k
MAWI15	MAWI [7, 15]	135m	40.8k	17.3k	1.7k
FB	Facebook [6, 22]	447m	n/a	n/a	n/a
UNI1	Univ. datacenter [4, 12]	18m	43.8k	1k	273
UNI2	Univ. datacenter [4, 12]	98m	7.7k	900	20

Table 1: Packet traces used in simulations

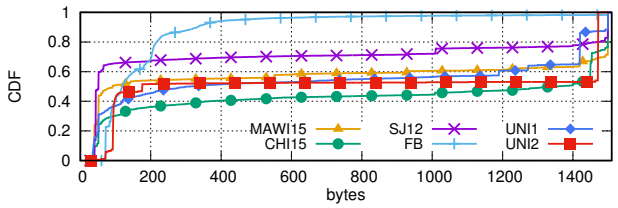

Figure 2: Packet size cumulative distribution.

4 MOTIVATING EXPERIMENTS

We implemented a trace-based simulator of the reference model presented in Sec. 2. We first use the simulator to measure the fraction of time a pipeline incurs into a data hazard, when processing packets with stateful operations spanning many clock cycles.

Traffic traces We fed our simulator with six traffic traces from both carrier networks (CHI15, SJ12, MAWI15) and datacenters (FB, UNI1, UNI2). Each trace presents different characteristics in terms of packet size (Fig. 2) and number of flows (Table 1).

CAIDA publishes 1h long traces captured from two backbone links in the US. We select CHI15 as a trace representative of typical conditions in CAIDA's monitored links, since the packet size and flows distributions are similar to the majority of the other CAIDA's traces published in recent years [3]. The packet size presents a bimodal distribution, 30% of packets have minimum size below 80B, while 50% have larger size close to 1500B. We select a second CAIDA's trace, SJ12, since it is representative of an unusual situation in the network when compared to CHI15. SJ12 has a prevalence of smaller packets, and a very large number of 5-tuple flows.

MAWI [15] publishes daily traces with 15 minutes of traffic captured from a transoceanic link between Japan and the US. We select MAWI15 since it is representative of typical traffic characteristics reported also by other MAWI traces.

FB includes packets collected from top-of-rack switches at a Facebook's datacenter cluster that serves web requests. This trace presents a significant predominance of small packets compared to other traces (80% have size < 200 bytes). Unfortunately, the traces provided by Facebook are the result of uniform sampling with a rate of 1:30k [6]. As a consequence, we were not able to precisely count the number of distinct flows. Hence, we use FB only to measure the effects of variable packet size, ignoring flow-level parallelism.

Finally, UNI1 and UNI2 are recorded from an edge switch at two different university datacenters. In UNI2 traffic belongs primarily to distributed file system applications, while in UNI1 traffic is split 60/40 between web services and file sharing applications [12].

Simulations Our simulator implements a pipeline taking as input (i) the number of pipeline's stages N, (ii) a definition of the flow granularity, and (iii) a packet trace.

Each stage is executed in 1 clock cycle. The first stage reads from the state, while the last one writes to it. N represents the the number of stages in the pipeline, i.e. the pipeline length, which in our case corresponds to the number of clock cycles in between a state read and write operations. The simulator finds a data hazard whenever the pipeline contains two packet headers whose processing requires access to the same portion of the state. Clearly, when $N = 1$, there is no risk of data hazards. Packets are read in chunks of 80B (as in RMT), hence taking 19 clock cycles to read 1 packet of 1500B. For $N \leq 18$, there is no risk of data hazard for 1500B packets. With smaller sizes the pipeline will experience shorter idle gaps, and hence a higher risk of data hazard. When considering flow-level parallelism, if two packet headers belonging to *distinct* flows are both in the pipeline, this does *not* generate a data hazard since they will access different portions of the state.

The simulator processes traffic in batches of 100k packets. For each batch, it outputs the fraction of data hazards (FDH), i.e, the number of clock cycles in which a data hazard was found divided by the total number of clock cycles needed to process 100k packets (which depends on the packet size). Finally, for each trace, we extract the 99th percentile from all the FDH values. As an example, if for a given trace the 99th percentile of the FDH is 0.3, it means that in the 99% of batches evaluated, the FDH was below 30%. To simulate 100% line rate speed, we accelerate the provided traces by feeding packets back-to-back, i.e. with no inter-packet gaps.

Fig. 3 (top) shows the results when packets are processed accessing a global state. As expected, different traces have different values of FDH. Those with a prevalence of larger packets have smaller FDH. E.g., FB with $N > 5$, produces a FDH of about 35% when accessing global state; the more favorable packet size distribution of CHI15 produces a FDH smaller than 10% even for $N = 30$.

The bottom part of the figure plots FDH values when taking into account flow parallelism. Using a finer flows granularity can greatly reduce the FDH. For instance, with SJ12 and a flow definition based on the destination IP address, FDH is about 24% with $N = 30$. When using the 5-tuple to access state, the FDH lowers to less than 3%. In the favorable case of CHI15, when taking into account flow parallelism, the FDH is about 1% even for stateful pipelines of up to 30 clock cycles.

5 APPROACH: MEMORY LOCKING

The FDH results can be read as an indication of the amount of time the pipeline should be stalled, in order to avoid data hazards. Stalling the pipeline can affect the system throughput, however, the cost of a stall could be potentially amortized during otherwise idle periods.

On the basis of this observation, we designed a locking scheme that stalls the pipeline whenever there is a new packet that would trigger a data hazard. In particular, if two packets that require access to the same portion of the memory arrive back-to-back, processing is paused for the second packet until the first one has updated the memory. To enable the amortization, we handle the arrival of packet bursts introducing queues and a scheduler to arbitrate access to the portion of the pipeline that needs access to state.

Fig. 4 shows the proposed design. The pipeline is preceded by Q queues, of length Q_{len}, and a scheduler. For each new packet, a flow key (FK) is extracted from its headers. A dispatcher stores the headers in the q-th queue, where $q = hash(FK)\ mod\ Q$. Since

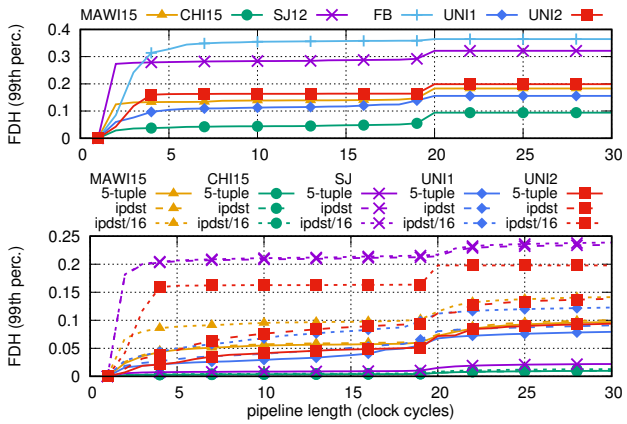

Figure 3: Fraction of data hazards (FDH) with increasing pipeline length. (top) presents the case of global state; (bottom) shows the result when accounting for flow parallelism.

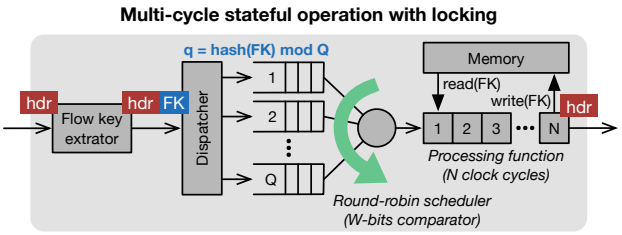

Figure 4: Locking scheme architecture.

packets belonging to a flow are assigned the same flow key, hence the same queue, per-flow packets ordering is preserved.

The scheduler admits packets in the processing pipeline by looking at each queue, and comparing the head-of-line FK with the at most N FKs currently traveling in the pipeline. A header is admitted only if its FK is *not* currently in the pipeline. We assume the scheduler is work-conserving, i.e., if at least one header can be served it will do so. As such, all non-empty queues should be compared at the same time. To avoid starvation, the scheduler serves queues in a round-robin fashion, i.e., with cyclic priority.

We assume a FK can have arbitrary length of FK_{len} bits. If $Q < 2^{FK_{len}}$, multiple flows will end up sharing the same queue. Such an event may generate head-of-line blocking, where all packets in a queue are held by the first one. The problem can be alleviated by adding more queues. However, this would increase the system cost in terms of memory needed to implement the queues. Furthermore, it could make it harder for the scheduler comparator to respect timing constraints. In fact, the work-conserving scheduler compares the FK from all queues' head-of-line at the same time, hence increasing the number of wires with i) the number of queues and ii) the number of bits to compare for each queue. For this reason, we simplify the implementation by reducing FK to a smaller space of W bits. The reduction is performed by the flow key extractor, which along with the full FK extends the headers with a field w.

We compute $w = hash(FK)\ mod\ W$. For example, with $W = 4$, the scheduler is able to distinguish among $2^4 = 16$ flows. If $W < FK_{len}$ there will be different flows colliding onto the same value w, impacting performance. The occurrence of collisions also depends on the hash function. In our experiments we use CRC16, which is a common choice in switching hardware architectures. We did not investigate the impact of other hash functions.

N	8	16	32
Area at 1 GHz	196 μm^2	929 μm^2	1560 μm^2
Min. delay	240 ps	360 ps	400 ps

Table 2: Area and minimum critical-path delay of one comparator in a 45 nm standard-cell library when $W = 4$.

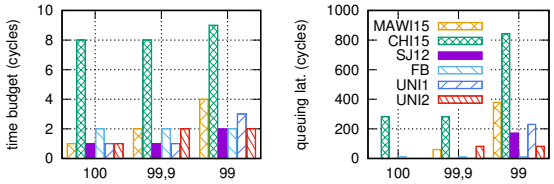

Figure 5: Per-packet maximum time budget (left) and 99th percentile of the queuing latency (right) when accessing global state, targeting 100%, 99.9% and 99% line rate throughput.

Notice that our design is not novel and different schemes may be applicable here. However, we are interested in understanding the feasibility, in principle, of such locking schemes, and in evaluating the impact they would have on the data plane performance.

5.1 Feasibility

We now provide a preliminary evaluation of the feasibility of the proposed design. The combinatorial logic complexity of the locking scheme is that of Q comparators, each one comparing the W bits of the head of the queue with each of the N stages of the pipeline. We have synthesized the comparators using the Synopsys Design Compiler [5] and a 45 nm standard-cell library. The VHDL code is available at [9]. Table 2 shows the area and minimum critical-path delay of one comparator when $W = 4$ and $N = 8, 16, 32$. In all the cases, timing constraints are easily met at 1 GHz, which is the clock frequency of state-of-the-art architectures such as RMT and Banzai. In terms of area overhead, when comparing our results to a 200 mm^2 chip as in [16, 24], we find that comparators area is negligible. For example, the area of 16 comparators ($Q = 16$) when $N = 16$ and $W = 4$, corresponds to the 0.01% of the total chip area.

The memory requirements to implement queues is $H_{len} \times Q \times Q_{len}$ bits, where H_{len} is the length in bits of the data path. With $H_{len} =$ 88 bytes (80 for the header and 8 for the metadata), $Q = 4$ and $Q_{len} = 100$, the locking scheme requires 35.2KB of memory for the queues. Approximately a 3.5% memory overhead compared with the memory of a MAT stage in RMT.

Finally, there is a silicon overhead associated to the N stages implementing the actual stateful processing function, which we do not discuss here. However, it is worth mentioning the RMT implementation result: 80% of chip area is due to memory (TCAMs and the IO/buffer/queue subsystem), and less than 20% area is due to logic. As a consequence, even if we did not test it, we are reasonably optimistic about the possibility to support more complex processing functions, which span many pipeline stages.

5.2 Evaluation

We extended the simulator described in Sec. 4 to model the proposed solution and evaluate it. The extended simulator takes the following parameters as input: flow granularity; pipeline length (N); queues number and size (Q and $Qlen$); comparator's bits number (W).

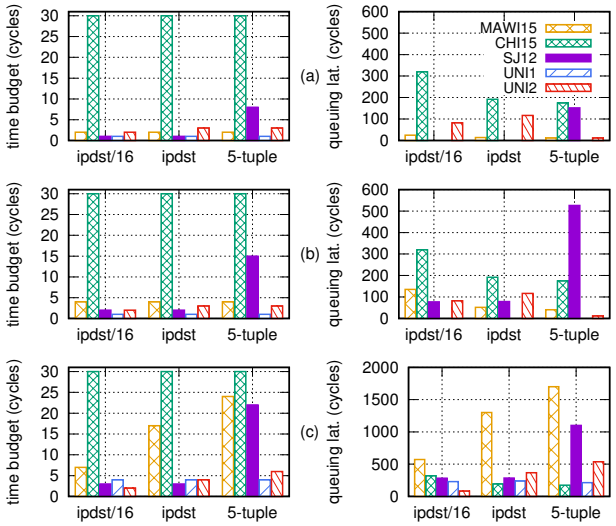

Figure 6: Per-packet maximum time budget (left) and 99th percentile of the queuing latency (right) when accessing state using different flow granularities, targeting 100% (a), 99.9% (b) and 99% (c) line rate throughput.

The aim of this round of simulations is twofold. First, we evaluate the biggest pipeline length (N) that achieves a target throughput. We consider the 100% of the line rate and slightly lower values, i.e., 99.9% and 99%, as viable targets. Second, we evaluate the *queuing latency* (QL) introduced by the locking scheme. The per-packet QL is the number of clock cycles a packet spends in one of the Q queues. It is a direct effect of employing a blocking architecture. When a packet finds an empty queue, its QL is 0.

We run the simulator using the traffic traces of Sec. 4. For the QL, we measure the 99th percentile among all per-packet QL values when processing a batch of 100k packets, finally we report the maximum among all batches for a given trace. The results are provided in [9], here we report few highlights. If not stated differently, the reported results are computed for $W = 4$ and $Q_{len} = 100$.

Packet size This set of simulations ignores flow-level parallelism, assuming that all the packets access the same state. Therefore, only the packet size distribution has an impact on the measured metrics. Since there is no flow distinction, we simulate a pipeline with a single queue ($Q = 1$). Fig. 5 shows the results. As expected, CHI15 has the bigger N, because of the larger mean packet size. The other traces, instead, do not benefit from the locking scheme, and can only provide the target throughput with a single clock cycle pipeline (2 cycles in the case of FB). Reducing the target throughput, from 100% to 99% of the line rate, provides limited benefits in terms pipeline length. In fact, only the MAWI15 and UNI1 traces could benefit a larger N of 4 and 3 cycles, respectively. In any case, the additional cycles come at the cost of increased QL. E.g., with CHI15, the locking scheme allows for $N = 8$, at the price of a 99th percentile QL of 282 cycles, when providing 100% throughput.

Flow parallelism The situation improves when considering per-flow state accesses. Fig. 6a shows N and QL when changing the granularity of the flow definition, using 4 queues ($Q = 4$). In the case of CHI15, N sensibly increases to 30 cycles: the largest N we tested in our simulations. For the most granular flow definition

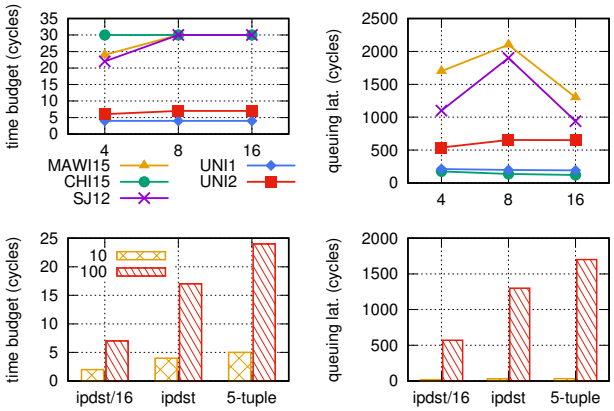

Figure 7: Per-packet maximum time budget (left) and 99th percentile of the queuing latency (right), when using a 5-tuple flow definition and targeting 99% line rate throughput. The top figures show the effect of different number of queues, the bottom ones the effect of different Q_{len} values on MAWI15.

(5-tuple), also SJ12 can use a longer pipeline with $N = 8$. Slightly reducing the target throughput to 99.9% (Fig. 6b), or 99% (Fig. 6c), provides further benefits. For instance, with a 5-tuple granularity, all the carrier network traces can have $N > 20$ when providing 99% throughput. Again, a larger N comes with the disadvantage of a longer queuing time. In the previously mentioned case, the 99% throughput is achieved with a QL 99th percentile of 1300 and 1700 cycles for SJ12 and MAWI15, respectively.

Queues The number and size of queues have an impact on N and QL. Fig. 7 (top) shows the effect of using varying Q when targeting a 99% throughput with 5-tuple flows. More queues provide additional space for packets, allowing the system to provide higher throughput without drops. E.g., with 8 queues all the carrier network traces can use up to $N = 30$, in which case packet's QL accordingly increases. Nonetheless, for a fixed value of N, adding queues can actually reduce the experienced QL by mitigating the head-of-line blocking.

Reducing Q_{len}, instead, reduces the time a packet can wait to be served. Thus, QL is greatly reduced, but at the cost of a much lower N: something particularly severe for traces with a small packet size. Fig. 7 (bottom) shows this effect for MAWI15 when providing 99% throughput with 5-tuple flows.

6 DISCUSSION

Our simulation results show it is possible to use a longer pipeline to perform consistent state operations, when taking into account actual traffic characteristics. Furthermore, they show that taking into account state access patterns, i.e., locking state accesses on a per-flow basis rather than globally, can increase the time budget to process a packet. These results are highlighted in Fig. 8, where the simulation's throughput results for real traffic traces are compared to a simulation of the worst case currently assumed in stateful data plane designs (only global states and all minimum size packets).

The ability to support a longer pipeline could potentially allow a data plane to provide more complex stateful operations, hence it could help in supporting a larger set of use cases. Also, having a larger per-packet time budget may reduce the technical challenges of

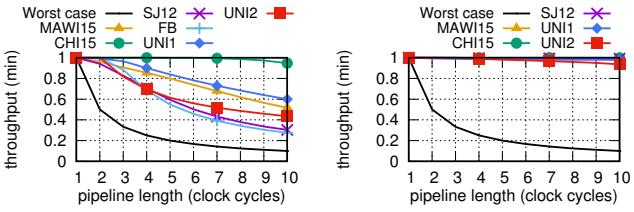

Figure 8: Maximum throughput when considering global state (left) and flow state for a 5-tuple granularity (right), compared to the worst case (global state and packets all minimum size).

designing a high-performance data plane, providing benefits such as reduced design costs or allowing designs to use cheaper components.

However, the length of the pipeline, the packet forwarding latency and the configuration of the locking scheme are dependent on the specific traffic properties. For instance, the packet size distribution and the number of the considered network flows play an important role. The traces used in our evaluation highlight this dependency. E.g., while CHI15 shows that a typical bimodal packet size distribution can be processed by a long pipeline, the other traces show that packet sizes play only part of the game. In fact, SJ12 can be processed by a longer pipeline than MAWI15, when considering a 5-tuple flow granularity (e.g., Fig. 6a), even if it has a less favorable packet size distribution (cf. Fig. 2). Here, the SJ12's large number of 5-tuple flows (cf. Table 1) is the reason.

Furthermore, notice that the flow definition, the required data plane operations complexity and the maximum tolerable forwarding latency are application-dependent variables. In practice, this means that some applications may be supported by pipelines of a given length, which can implement the required stateful operations, while others may require longer (or shorter) pipelines. That is, the simulation results and the variability of the applications' requirements make it clear that there is no one-size-fits-all solution. This justifies designers in taking conservative decisions. I.e., data plane pipelines that provide limited stateful operations in order to guarantee performance, independently from the traffic characteristics [24].

Nonetheless, our work suggests there could be an opportunity to rethink stateful data plane designs and abstractions. E.g., the locking architecture we described in this paper proved to be effective while requiring little resources. This could open the space for designing different (fixed) versions of a data plane pipeline. Each version would target a particular deployment environment, e.g., carrier vs. data center network, i.e. characterized by the expected traffic properties, and could provide different performance trade-offs.

In perspective, a programmer could use the expected traffic characteristics, and her application properties such as the flow definition, to configure the pipeline stateful operations. Notice that this configuration step would be in line with the RMT approach, where the number, type and size of MATs can be configured based on the application's needs. Furthermore, we believe this point could be even more relevant when considering pipelines implemented using different technologies, such as FPGAs [25] and SmartNICs [8], which allow for a deeper reconfigurability.

7 CONCLUSION

We showed that taking into account traffic characteristics, in particular when using a flow-based model to access a data plane's state, can increase the packet processing time budget. This observation has a direct impact on the design of high-performance switching pipelines, where the time budget relates to the number of clock cycles that can be used to perform consistent data plane's state modifications. Furthermore, we presented a sketch of a locking scheme that allows the switching pipeline to take advantage of the additional per-packet time without harming state consistency. While the presented locking scheme is not novel, it shows such an approach is both effective and applicable with little overheads. The code used for the simulations is available at [9].

A number of related issues could not find space in this paper. For instance, we did not provide security considerations about the proposed blocking architecture, and we did not elaborate on which stateful operations could be implemented with the additional time budget. While we believe covering such issues is important, we also consider them not strictly required for the contribution of this paper. Nonetheless, such issues are included in our future work.

REFERENCES

[1] The CAIDA UCSD anonymized internet traces - Chicago 2015-02-19. http://www.caida.org/data/passive/passive_2015_dataset.xml.

[2] The CAIDA UCSD anonymized internet traces - San Jose 2012-11-15. http://www.caida.org/data/passive/passive_2012_dataset.xml.

[3] The CAIDA UCSD statistical information for the CAIDA anonymized internet traces. http://www.caida.org/data/passive/passive_trace_statistics.xml.

[4] Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

[5] Design Compiler - Synopsys. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[6] FBFlow dataset - cluster B, howpublished = https://www.facebook.com/network-analytics.

[7] MAWILab traffic trace - 2015-07-20. http://www.fukuda-lab.org/mawilab/v1.1/2015/07/20/20150720.html.

[8] Netronome AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[9] Simulator code and results. https://github.com/ccascone/opp-sim.

[10] XPliant ethernet switch product family. http://www.cavium.com/XPliant-Ethernet-Switch-ProductFamily.html.

[11] M. Alizadeh, et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM 2014*.

[12] T. Benson, et al. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM IMC 2010*.

[13] G. Bianchi, et al. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *ACM SIGCOMM CCR*, 44(2), 2014.

[14] P. Bosshart, et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013*.

[15] R. Fontugne, et al. MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *ACM CoNEXT 2010*.

[16] G. Gibb, et al. Design principles for packet parsers. In *ACM/IEEE ANCS 2013*.

[17] A. Kaufmann, et al. FlexNIC: rethinking network DMA. In *USENIX HotOS 2015*.

[18] N. McKeown, et al. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.

[19] M. Moshref, et al. Flow-level state transition as a new switch primitive for SDN. In *ACM HotSDN 2014*.

[20] A. Panda, et al. Verifying reachability in networks with mutable datapaths. In *USENIX NSDI 2017*.

[21] S. Pontarelli, et al. Stateful Openflow: Hardware proof of concept. In *IEEE HPSR 2015*.

[22] A. Roy, et al. Inside the social network's (datacenter) network. In *ACM SIGCOMM 2015*.

[23] N. K. Sharma, et al. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI 2017*.

[24] A. Sivaraman, et al. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM 2016*.

[25] N. Zilberman, et al. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5), 2014.