FlowBlaze: Stateful Packet Processing in Hardware

Salvatore Pontarelli, *Axbryd/CNIT;* Roberto Bifulco, *NEC Laboratories Europe;*
Marco Bonola, *Axbryd/CNIT;* Carmelo Cascone, *Open Networking Foundation;*
Marco Spaziani and Valerio Bruschi, *CNIT/University of Rome Tor Vergata;*
Davide Sanvito, *Politecnico di Milano;* Giuseppe Siracusano, *NEC Laboratories Europe;*
Antonio Capone, *Politecnico di Milano;*
Michio Honda and Felipe Huici, *NEC Laboratories Europe;*
Giuseppe Bianchi, *CNIT/University of Rome Tor Vergata*

https://www.usenix.org/conference/nsdi19/presentation/pontarelli

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

FlowBlaze: Stateful Packet Processing in Hardware

Salvatore Pontarelli[1,2], Roberto Bifulco[3], Marco Bonola[1,2], Carmelo Cascone[4], Marco Spaziani[2,5], Valerio Bruschi[2,5], Davide Sanvito[6], Giuseppe Siracusano[3], Antonio Capone[6], Michio Honda[3], Felipe Huici[3] and Giuseppe Bianchi[2,5]

[1]Axbryd, [2]CNIT, [3]NEC Laboratories Europe, [4]Open Networking Foundation, [5]University of Rome Tor Vergata, [6]Politecnico di Milano

Abstract

While programmable NICs allow for better scalability to handle growing network workloads, providing an expressive yet simple abstraction to program stateful network functions in hardware remains a research challenge. We address the problem with FlowBlaze, an open abstraction for building stateful packet processing functions in hardware. The abstraction is based on Extended Finite State Machines and introduces the explicit definition of flow state, allowing FlowBlaze to leverage flow-level parallelism. FlowBlaze is expressive, supporting a wide range of complex network functions, and easy to use, hiding low-level hardware implementation issues from the programmer. Our implementation of FlowBlaze on a NetFPGA SmartNIC achieves very low latency (in the order of a few microseconds), consumes relatively little power, can hold per-flow state for hundreds of thousands of flows, and yields speeds of 40 Gb/s, allowing for even higher speeds on newer FPGA models. Both hardware and software implementations of FlowBlaze are publicly available.

1 Introduction

Network infrastructures need a continuously evolving set of network functions to be operated reliably [45, 35]; NAT, path and server load balancing, traffic shaping, security functions such as access control, and DoS protection are just a few examples. Given the flexibility of modern networks and the need to continuously support new applications, operators have turned to pure software implementations for such functions, which have a number of benefits, including programmability and ease of management [8].

However, while essential to network operations, network functions introduce overheads. Most notably, they increase network packets' end-to-end delay, since they are on the path of network flows, and increase the overall cost of running the infrastructure, needing additional computation resources, i.e., CPU cores. These overheads become particularly critical in low latency fabrics such as those of modern datacenters [50]. For instance, server-to-server communication delays are expected to be in the order of a few tens of µs in a cloud datacenter [29]. For comparison, it could take tens of µs for packets to go from a NIC, over the PCIe bus, to a CPU that executes a network function and back to the NIC.

To address this issue, past years have seen the introduction of efficient programmable network devices that can offload network packets processing from the CPU. For example, Microsoft deployed FPGA-based SmartNICs in their datacenters [24]. These devices save CPU usage and reduce the amount of traffic on a server's PCIe bus, improving a network function's packet processing latency by more than an order of magnitude. As a downside, programming a SmartNIC to support a new network function requires hardware design expertise. While a tech giant can build and assign a dedicated team to the task [24], this is usually not the case for a large majority of companies, e.g., smaller cloud or network operators. As a result, recent network programming abstractions, such as P4 [13] have the explicity goal of simplifying the programming of FPGA-based network devices [60, 2]. However, they have limitations in describing network functions that need to keep per-flow state [24, 58], a common requirement in the world of network functions.

In this paper we address these shortcomings by introducing FlowBlaze, an abstraction that extends match-action languages such as P4 or Microsoft's GFT [24] to simplify the description of a large set of L2-L4 stateful functions, while making them amenable to (line-rate) implementations on FPGA-based SmartNICs. To benefit the community at large, we build FlowBlaze on open SmartNIC technology (NetFPGA [65]), and provide our hardware and software implementations as open source. Our contributions are:

- The FlowBlaze abstraction. FlowBlaze adapts match-action tables to describe the evolution of a network flow's state using Extended Finite State Machines (EFSM).
- A hardware implementation of FlowBlaze on the NetFPGA SUME card that can forward packets at 40Gb/s line rate while keeping state for hundreds of thousands of flows, consuming relatively little power, and providing

packet processing latency of less than 1 µs.
- A comprehensive analysis of FlowBlaze' expresiveness through the implementation of over 10 different use cases including stateful firewalls, load balancers, NATs, traffic policers and SYN flood detection and mitigation.
- Two different high performance software implementations (for mSwitch [31] and the Linux kernel) of FlowBlaze. Such implementations can be used to handle VM-to-VM communications; to help implement fail-over in software scenarios, e.g., during hardware maintenance; to support end systems that do not have a SmartNIC available; or to implement network functions that are deployed in less performance-demanding scenarios.

FlowBlaze implementations, documentation and additional results are available at [25].

2 Requirements and state-of-the-art

Our goal is to provide a system that allows a programmer with little hardware design expertise to quickly implement, or update, stateless and stateful packet processing functions at high speed, on FPGA-based SmartNICs. It should be noted that our focus is on functions that operate on packet headers, generally at L2-L4 of the network stack, such as firewalls, L4 load balancers and NATs. These are common building blocks for many other functions, and can often be completely executed within a NIC. Functions that operate on packet payloads such as DPIs and L7 firewalls are out of the scope of this work. More formally, we target the following requirements:

- **R1: High Performance**: support of network functions that achieve throughput in the range of 40-100Gb/s while providing per-packet processing latency of at most a few µs.
- **R2: State Scalability**: support for functions that operate on fine-grained per-flow state and the ability to store per-flow state for large numbers of network flows (e.g., up to several 100Ks). The number of flows should not affect the processing latency.
- **R3: Easy to Use:** allow a programmer to focus solely on implementing the functionality needed and not get bogged down in tricky, time-consuming hardware performance optimizations. Further, hardware constraints should have reasonably little impact on application design, and the programmer should need little to no hardware design expertise to implement a function.
- **R4: Expressiveness**: the system's programming abstraction should allow users to describe a large range of potentially *stateful* functions, including complex ones (e.g., anomaly detection, connection tracking, etc.).

2.1 Existing systems

Taking these requirements into account, we review the state of the art and find that existing systems only meet these requirements partially, for an FPGA target (See Table 1). In effect, we can split previous approaches in two categories:

	High Perf	State Scal	Ease	Expresiv
General programming frameworks				
HDL	√	√	×	√
HLS [52]	×	√	√	√
ClickNP [42]	-	√	-	√
Match-action abstractions				
P4 [13]	-	-	√	-
Domino [58]	√	√	-	-
OpenState [11]	√	√	√	×
FAST [51]	×	√	√	×

Table 1: Qualitative comparison of stateful abstractions. A dash means a requirement is only partly met.

General programming frameworks are solutions that rely entirely on FPGA re-programmability features to implement new functions and therefore focus on languages and frameworks to simplify FPGA programming. Here we include Hardware Description Languages (HDLs) and High Level Synthesis (HLS) systems.

HDLs such as Verilog are a low-level programming method for FPGAs, requiring extensive hardware design expertise. HLS systems like those based on OpenCL, can improve ease of use (R3) by adopting high-level languages. However, hardware expertise is still needed since the code has to be properly designed and annotated to ensure the synthesis process succeeds in providing a high-performance implementation [52]. For example, ClickNP [42] may require the programmer of a functional element to apply specific hardware-related optimizations, when the compiler fails to apply its automated optimization [42]. Further, updating a function requires a new synthesis and flashing of the FPGA design, a process that takes hours.

Match-action abstractions are based on the match-action tables (MATs) model. MATs are an effective and widely applied tool to describe network functions as a pipeline composed of a parser and a variable number of match and action blocks. The parser and the actions logic are generally rather stable and are therefore programmed at configuration time, while the match table's entries are inserted at runtime and can be used to change the implementation of functions on-the-fly.

MATs are a good tool to describe L2-L4 network functions [47], but currently available MAT abstractions only support *stateless* functions so that a programmer cannot specify functions where the processing of a packet should depend on a previously received packet. Older versions of the P4 language, which targets a MAT model, left the implementation of state-related constructs such as registers, to platform-specific features, which reduces portability and complicates the work of a programmer. As a matter of fact, solutions that use P4 to implement FPGA-based network functions require the programmer to provide stateful func-

tions using HDLs [2, 60].

A step forward in this direction was first provided by Domino [58], which extends the match-action model to include, in the action blocks, small and fast registers that can keep state. These registers can be accessed during the processing of any packet handled by the action block, thus providing a global state access model. A similar solution is also supported by the @atomic construct in the latest P4 language version, i.e., P4-16 [20] (the previous version was P4-14 [19]). Unfortunately, the global state model provides little information for hardware implementations to optimize state operations. Thus, Domino-like solutions are designed to address a worst-case access pattern, which leads to very constrained state update functionality (R4).

Do we meet all four requirements at once? The widespread application of MATs suggests that they could be a good starting point for providing an effective abstraction to describe network functions, and works like Domino already point towards a direction that extends MATs to support stateful functions. Thus, our problem boils down to supporting per-flow states in a match-action abstraction.

Here, we begin by making the observation that many network functions already partition their state on a per-flow basis [54]. This means that programmers are familiar with the concept of flow state, and that this inherent, per-flow parallelism can be leveraged to optimize state operations and memory accesses in a hardware implementation.

Can flow state become a first class citizen? Admittedly, FAST [51] and OpenState [11] follow this direction to provide stateful packet processing abstractions, explicitly defining flow state. In both cases, packets are grouped in programmer-defined network flows (like in MATs), and flows are associated with a Finite State Machine (FSM). Programmers then define the FSMs' transition tables which are used to update the corresponding FSMs' states as packets are processed. Following this line of reasoning, it would seem that FSMs would be a good choice to transform a stateless MAT into a stateful element. Related work has shown FSMs to be programmer-friendly, allowing for the definition of a host of packet forwarding behavior [40, 62] (R3, R4). Further, a FSM can be efficiently represented by its transitions table, which can be implemented in a straightforward way in hardware since it is functionally equivalent to a MAT (R1).

Unfortunately, FSMs are not scalable (R2). Briefly, an FSM is described by the set of states S, inputs I, outputs O and by the transition relation $T : S \times I \rightarrow S \times O$. Since FSMs need to explicitly define all the possible states $s_i \in S$ the system can be in, this may lead to a phenomenon known as *state explosion* [34]; the next section explains how FlowBlaze solves this issue.

3 FlowBlaze Design

To keep most of the good properties of FSMs while providing a scalable abstraction, we resort to Extended Finite State Machines (EFSMs) [4]. An EFSM extends the FSM model by introducing: (i) variables D to describe the state; (ii) enabling functions F on such variables to trigger transitions ($f_i : D \leftarrow \{0,1\}, f_i \in F$); and (iii) update functions for the variable values ($u_i : D \leftarrow D, u_i \in U$). The transition relation of an EFSM is expressed as $T : S \times F \times I \rightarrow S \times U \times O$.

Example. Figure 1 shows the EFSM representation of an application that identifies whether a single flow f_1 (e.g., identified by IP destination) is large by marking all packets after the 100th one. Using a conventional graph representation, the nodes of Figure 1 are states, while edges represent transitions. Each node is named using the corresponding state label. Transitions are marked with the quadruple {*enabling functions*, *event*, *update functions*, *output*}. Enabling and update functions operate on the variable $D(f_1)$, which is the variable of D we selected to store a flow's number of packets. The event *pkt(f_1)* represents the reception of any packet belonging to a flow f_1. Finally, the outputs *mark* and *fwd* are high-level descriptions of a packet header rewriting action and a generic forwarding action. The dashed line shows a transition triggered by a timeout event (e.g., an idle timeout), which brings the EFSM back to its starting state.

3.1 The FlowBlaze Abstraction

While adopting EFSMs partly helps in dealing with state explosion, we still need to adapt them to ensure an efficient hardware implementation. We need to address two issues:

- **State scalability**: standard EFSMs would require a separate transition table (i.e., the EFSM's description), for *each* flow in the system.
- **Flow parallelism**: an EFSM state definition does not include the concept of flow state, which we need in order to leverage flow-level parallelism.

To address the above issues FlowBlaze uses EFSM *definitions* to describe the behavior of a group of flows. Here, the observation is that often many flows share the same forwarding behavior [39], even if each flow, at any given point in time, may be in a different state of such EFSM. For example, in Figure 1 we specify an EFSM for a single flow, but we usually want to apply the same EFSM to all the other flows seen by our function, so to mark *any* flow with more than 100 packets. In other words, the forwarding logic is the same for all flows, but they are actually associated with different EFSM *instances*.

State types The introduction of such instances has an immediate side-effect: there is no way for two different instances to read or write each other's states. For example, we would be unable to extend the EFSM of Fig. 1 to also count the total number of flows that sent more than 100 packets.

We address the issue by introducing the notion of *global state*, which can be read and modified by all the EFSM instances generated from the same EFSM definition. Formally, the FlowBlaze abstraction divides the EFSM variable space

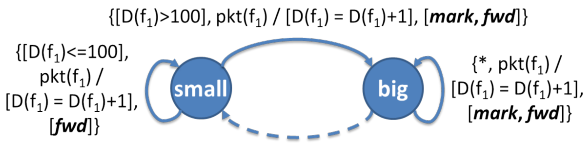

Figure 1: EFSM description of an application that identifies flows generating more than 100 packets

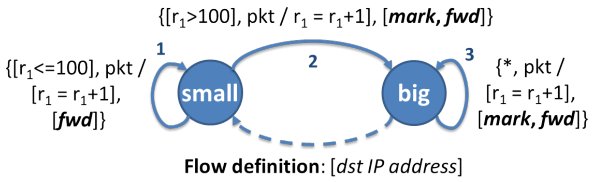

Figure 2: Description of the application of Fig. 1 using a generic flow definition.

D in two parts: the global registers $g_i \in G$ that are part of the global state; and the flow registers $r_i \in R$, which, together with the state label, constitute the flow state. The global state exists for the entire lifetime of the system and accesses to the global state are serialized, similar to [58]. Flow state, on the other hand, only exists while its corresponding flow does, and accesses to it are serialized for packets *within* a flow, i.e., when processing packets belonging to different flows it is possible modify their respective flow states in parallel.

To illustrate, Figure 2 extends the example of Figure 1 to describe the same application but this time using an EFSM to identify all large flows, with a flow defined as a set of packets with the same destination IP address. In this case, each flow is associated with an EFSM instance that has its own current state ("small" or "big") and a variable r_1. Notice that in Figure 2 variables are no longer accessed using a flow id (in Fig. 1 we used $D(f_1)$), since each packet is now associated with its corresponding flow's EFSM instance. Similarly, the packet reception event does not specify the flow the packet belongs to anymore, since that information is captured already by the generic flow definition associated with the EFSM. Table 2 shows the transition table needed to implement this function; we will give a full explanation of it and of how to program FlowBlaze later in this section.

Composition In a match-action abstraction there is usually a pipeline of MATs. Similarly, FlowBlaze allows for the pipelining of EFSMs, which is equivalent to their sequential composition [40]. That is, in the most general case a network function is described by an ordered list of EFSM definitions, where the output of the EFSM i can be used as input for EFSM $i+1$. Each EFSM definition has its own scope for the flow and global states: an EFSM definition's global state is only accessible by that EFSM's instances.

To move information between two sequential EFSMs, FlowBlaze associates *packet state*, i.e., metadata, with each packet. Such state is created when a packet is received, and deleted once its processing is completed. For example, after the EFSM of Figure 2 one could add another EFSM which uses metadata tagged by the first EFSM to classify packets differently, based on whether they belong to a large or a small flow.

#	Cond.	Event	S	Nxt S	Update	Pkt act.
1	$r_1 <= 100$	pkt	sml	sml	$r_1 = r_1 + 1$	fwd
2	$r_1 > 100$	pkt	sml	big	$r_1 = r_1 + 1$	mrk, fwd
3	*	pkt	big	big	$r_1 = r_1 + 1$	mrk, fwd

Table 2: Transition table for the example of Fig. 2.

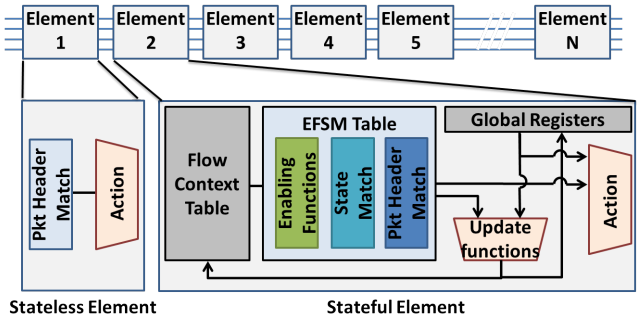

Figure 3: FlowBlaze machine model

3.2 Machine Model

Having described the FlowBlaze abstraction, we now proceed to show how this abstraction is formalized to a machine model that can be implemented in hardware.

More specifically, FlowBlaze's machine model (see Figure 3) extends the MATs pipeline model described by RMT [14]. Like in RMT, FlowBlaze packet headers (including packet metadata) are processed through the pipeline's elements to define the forwarding actions. Each element can be either stateless or stateful. A stateless element is a MAT, similar to the ones employed in RMT[1]. Unlike RMT, a stateful element implements an EFSM definition. As a result, a pipeline can combine both stateless and stateful elements.

The architecture of a stateful element has two notable differences from a MAT: (1) such an element has a *Flow Context Table* before the usual match part, and (2) the element splits the actions into (state) update functions and packet actions. In greater detail, as shown in Figure 3 (the box labeled "Stateful Element"), packet processing involves the following sequential steps:

1. Flow Context Table. When a packet header enters the element, it is associated with a corresponding flow context. The context is extracted from the Flow Context Table using, as search key, a list of header fields (e.g., the TCP/UDP 4-tuple, optionally in conjunction with the packet's metadata). The search key is specified at configuration time and corresponds to FlowBlaze's EFSM flow definition. The context, i.e., a table's entry, includes a state label s and an array of

[1]FlowBlaze assumes packets headers are already parsed when passed to the pipeline, taking advantage of RMT-like programmable packet parsing [28] and reconfigurable match tables [14].

registers $\vec{R} = \{r_0, r_1, ..., r_{(k-1)}\}$. Flow contexts are also associated with hard and idle timeouts. If no context is found for a given key, a default context is used (i.e., with all values set to 0). A single flow context identifies an EFSM instance.

2. EFSM Table. The packet's header and metadata, plus the extracted flow context, are passed to the *EFSM table*. Such table is an extension of a traditional MAT, which in addition to supporting matching on the packer header fields, can also match on the state label s and evaluate enabling functions. An enabling function can be specified as a logical AND of up to m arithmetic comparisons ($\vec{C} = \{c_0, c_1, ..., c_{(m-1)}\}$). The comparisons' operands can be selected among any combination of flow registers and packet header fields. For each entry in the table, a programmer can specify (i) packet modification and forwarding operations, (ii) the next state label s, and (iii) a list of instructions to update the flow context registers $\vec{R}$ and the global registers $\vec{G} = \{G_0, G_1, ..., G_{(h-1)}\}$. In short, the EFSM table acts as the state machine's transition table.

3. Update Functions. The header and metadata, the update instructions, and the new value of the state label are passed to the *update functions block*. The block performs the required update instructions to update the values stored in both the flow context registers $\vec{R}$ and global registers $\vec{G}$. Such instructions can range from simple integer sums, for instance to update the value of a register representing a packet or byte counter, to more complex ones, e.g., multiplications, depending on the specific implementation and target performance.

4. Action. Like in a MAT, this block applies actions on the packet header. In contrast to a MAT, the values of the flow context registers as well as those of the global registers can also be used (e.g., to rewrite some packet header fields).

3.3 FlowBlaze Programming

FlowBlaze is deployed bump-in-the-wire, in the NIC, and its programming is similar to programming a P4 device. At configuration time, the programmer has to define the parser, the match fields for MATs and EFSM transition tables, and the actions, which now include also the state update functions. Like in [2], changing these components requires a new synthesis of the FPGA design. Luckily, these are the parts of a function that change less frequently [23].

At runtime, the programmer defines the logic of her network functions by configuring the flow definition for the stateful elements, selecting a subset of the parsed header fields, and writing the required entries in the MATs and EFSM transition tables. This is analogous to the runtime programming of P4 and OpenFlow devices. In fact, we extend the OpenFlow protocol to write such entries from a python-based RYU OpenFlow controller [25].

One thing worth highlighting is that the programmer can quickly experiment and update her functions logic on-the-fly, since programming the network functions logic is as quick as writing entries to tables, unlike other solutions that need a new synthesis and flashing of the FPGA design [42].

Use case	Entries	Reg.
Server Load Balancer	2,2	0+1, 1
UDP Stateful Firewall	5	0
Port Knocking Firewall	6	0
Flowlet load balancer	2, 4, 9	1, 0+2, 0+4
Traffic Policer	4	2+2
Big Flow Detector	3	1
SYN flood Detection and Mitigation	4	1
TCP optimistic ACK detection	8	3
TCP super spreader detection	8	1
Dynamic NAT	3, 4	1+1, 2
vEPC subscriber's quota verification	9	1
Switch Paxos Coordinator	1	0+1
Switch Paxos Acceptor	3	3+1
In-network KVS cache	6	2

Table 3: Implemented use cases. The entries column reports the number of EFSM table entries needed by a stateful element, with each comma-separated number representing a different element. The registers column lists the number of flow+global registers for each element. More in the Appendix.

3.4 Expressiveness: A Case Study

To demonstrate FlowBlaze's expressiveness we have implemented a large set of network functions ranging from NATs to load balancers and anomaly detection, to name a few (full listings in Table 3). For each we state the number of stateful elements, and the respective number of entries and registers required by FlowBlaze to support them. The different use cases can be combined to provide more complex functions.

Beyond these applications, and to provide a better understanding of the level of complexity that our system can support, we now show an example of a FlowBlaze configuration that implements a TCP connection tracking function (Figure 4). Connection tracking is required by network operators to protect their networks from a number of attacks [33], but its implementation is fairly complex and challenging (e.g., it requires per-packet flow state updates to check TCP sequence numbers, and we cannot know, at least not right away, whether a packet we see is actually delivered or lost).

Assuming a parser can extract the relevant TCP header fields [28], FlowBlaze allows us to implement the function using two stateful elements. The first element has 7 transitions and tracks the TCP connection establishment, i.e., the three-way handshake, and computes receive window (RCW) boundaries. A connection is identified with a bi-directional 4-tuple network flow. In particular, the flow key specification is defined as *biflow*, meaning that both directions of the flow will be associated with the same flow context. The flow context contains the IP address of the initiator of the connection (r_0), the last acknowledged sequence numbers (SEQs) (r_1, r_2), the last seen SEQs (r_3, r_4), and the RCW for each direction (r_5, r_6). A packet is sent to the second element when a connection is in the *estbl* state, in which case the left and right boundaries of the RCW are computed and copied

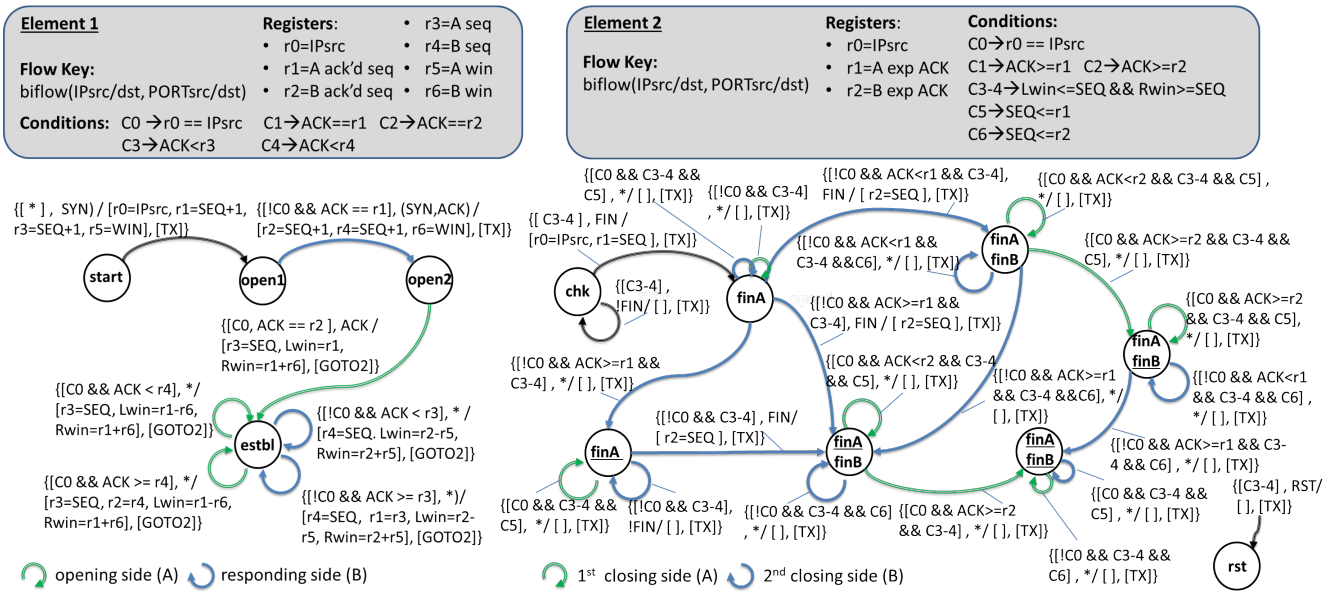

Figure 4: A complex, TCP connection tracking use case to show FlowBlaze's expressiveness. All of element 2's states can transition to the rst state. Transitions triggered by timeouts are omitted for clarity.

into the packet's metadata. The second element has 29 transitions and checks if the packet's SEQ is within the RCW boundaries and handles connection termination. The flow context contains the IP source of the connection termination initiator (r_0) and the last expected ACK number for each direction (r_1, r_2). Since connection terminations do not happen at packet processing speed, and since it is safe to keep state for terminated connections for a short time, we leave it to software (e.g., running on a CPU), to clean up the state of terminated connections: the software reads the terminated connections from element 2 and clears the state for those connections from element 1.

It is worth pointing out that making a few assumptions about the implemented network functions can seriously reduce the required FlowBlaze resources. For instance, we can reduce the number of registers from 7 to 4 in element 1 by assuming that one side of the connection is trusted [30], e.g., because we control the TCP stacks of those machines. Likewise, we could simplify the implementation assuming a fixed size for the receive window, as done in some hardware firewall implementations [55]. Finally, we could easily change the implementation behavior for SEQ checking from *window shifting* to *window advancing* by simply introducing an additional condition (i.e., $newSEQ > storedSEQ$) to the transitions that update the stored SEQ in element 1.

4 Hardware Design and Implementation

In this section we present FlowBlaze's hardware design, mapping the abstraction and machine model described in the previous section to an actual hardware implementation. Note that we focus mainly on the architecture of a stateful element, since that is where most of the research contribution in this section lies (we leave out details about packet header parsing). We begin by giving an overview of the stateful element's architecture, and follow that by a description of the issues we encountered and how we addressed them.

4.1 Stateful Element Architecture

FlowBlaze's hardware design extends a state-of-the-art packet processing pipeline by introducing stateful elements. As mentioned in Section 2, one of the advantages of selecting an EFSM-based abstraction is that the EFSM's transition table can be directly mapped to a MAT, which constitutes the starting point for our stateful element architecture (see Fig. 5): the TCAM, instruction RAM and packet header action blocks are components of a regular MAT implementation; the remaining blocks are required to implement the FlowBlaze abstraction.

In greater detail, the stateful element works as a pipeline. A packet header is received and hashed by the key extractor, which is configured to generate a flow key according to the provided EFSM's flow definition, and first handled by the scheduler block. This block uses the hash to place the header in one of its queues, guaranteeing no per-flow re-ordering, and then serves headers from the queues with a scheduling policy that provides per-flow state access consistency.

A scheduled header is then used to look up the corresponding flow state in the Flow Context Table. The state includes the state label s and the registers $\vec{r}$, values which are then fed as input to the conditions evaluation block. This block selects the operands for the conditions from the registers, the header fields and/or a constant value using a crossbar, and outputs the result to a vector $\vec{c}$, which, together with the state label s and the header fields, is used to perform a look-up in the EFSM table. The result of the look-up is the instruc-

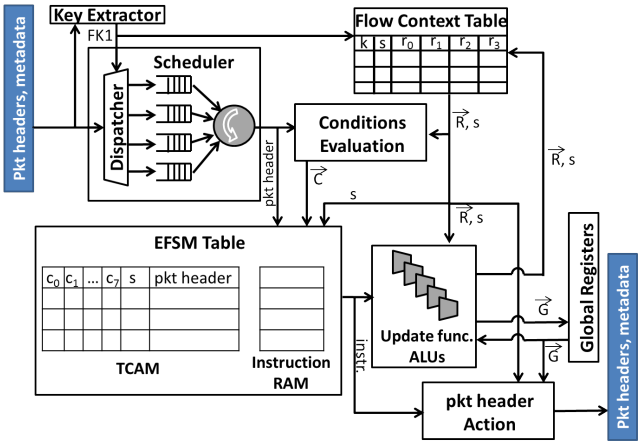

Figure 5: FlowBlaze's stateful element architecture.

tions that should be executed by the update function's ALUs and the packet header action block. These two blocks perform their operations in parallel: the former updates the flow state in the Flow Context Table and the value of the global registers, while the latter modifies the header fields.

Having given an overview of the architecture, we dedicate the rest of the section to describing how we solve issues to do with the scalability of the Flow Context Table and with guaranteeing consistency.

4.2 Scalability of Flow Context Tables

In order to implement the flow-oriented addressing required by the forwarding tables and by FlowBlaze's Flow Context Table we need to rely on optimized hash tables. A typical solution in this space is to use cuckoo hash tables [63] which support higher load factors, thereby improving SRAM usage efficiency. However, an entry insertion in a cuckoo hash table may actually need multiple operations when there is a hash collision; this makes insertion times variable and potentially long for a loaded table, severely impacting performance. While current designs usually perform entry insertion and collision handling in the device's control plane [24, 49], in FlowBlaze we need to handle entry insertions in the data plane while guaranteeing quick and constant insertion times[2]. FlowBlaze solves the issue by implementing the cuckoo insertion logic completely in hardware and extending the hash table with a small stash memory to hold entries waiting for insertion. The stash allows FlowBlaze to hide the variable insertion time of a cuckoo hash. We implement a four-choices cuckoo hash table that offers a 97% load factor [22, 26], using a dual port (read-after-write) SRAM to support two accesses in the same clock cycle for concurrent read and write operations.

The key extractor of Figure 5 generates the four hash keys required to address the table, and the table itself is coupled with a stash memory that can host 8 entries [41]. In contrast to a typical cuckoo with stash, a new entry is always inserted first in the stash, which guarantees constant insertion time (1 cycle). In parallel, the insertion logic moves entries from the stash to the hash table and operates as a regular cuckoo+stash implementation. The insertion logic can execute an entry insertion or a movement (needed to resolve collisions) per cycle, in about 6.4ns in our implementation. It is worth noting that while a very loaded table can cause the number of movements to grow by as much as 100x [26], in section 5 we empirically show that our 8-entry stash memory is enough to avoid packet losses for the rather large network traces we test against. The observations here is that unlike state write operations which happen for each packet arrival, an insertion happens only when a new network flow starts, a smaller and thus manageable rate.

Handling Corner Cases Despite these mechanisms, a Flow Context Table (and its stash) may exceptionally become full. At this stage, the right strategy is dependent on the network function being run (e.g., for a stateful firewall the right approach might be to reject any new connections).

FlowBlaze provides a programmer with the ability to implement the logic to handle such cases. When a packet belonging to a new flow is received, FlowBlaze checks, in addition to the look-up in the Flow Context Table, the table's occupancy level. If the table is full, a flag is set in the packet metadata, essentially indicating that there is no more space to save flow state for this flow. This flag can then be matched in the EFSM table, allowing the programmer to program the state machine to handle the case, e.g., by dropping the packet (and the flow) or by sending the packet out for further processing, e.g., in software. Further, when a table is full, FlowBlaze provides the option to install a new flow state entry by replacing an existing one. This is handled with a configurable eviction policy: the insertion logic can be configured to read the entries' flow register R_0 and to select for eviction the entry with the highest (or lowest) value. That is, a network function's FSM can use R_0 to enforce a custom eviction logic. For example, R_0 can be used to store the timestamp of the last seen packet in order to implement a logic that evicts the least active flows; alternatively, R_0 could store a packet counter to evict the smallest (or biggest) flow.

4.3 Guaranteeing Flow State Consistency

Flow-oriented memory addressing implies that a memory location is accessed only when a given flow's packet is processed. As a result, a given memory location is accessed at a rate that may be just a fraction of the overall packet processing rate (i.e., one access per cycle), and enables read/modify/write operations that span multiple clock cycles. This feature, however, introduces a state consistency problem, since a memory location's access times may vary depending on the traffic pattern, potentially leading to a concurrent read/write

[2]For reference, the device's control plane can insert/modify entries at a speed that is usually 1000-10000x slower than the packet processing rate. Even in cases where such a slow update rate is tolerable, consistency problems for the state and scalability concerns for the control plane arise [49].

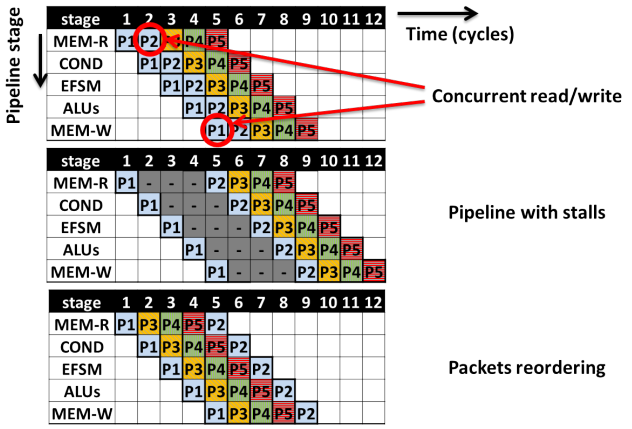

Figure 6: Stateful element scheduling scenarios. P1 (Packet 1) and P2 belong to the same flow and use the same flow context. P3, P4 and P5 belong to different flows and can concurrently access memory.

of the same location. For example, when two packet headers that access the same flow context entry are processed in short sequence, the second packet's flow state read may happen before the first packet's flow state update has been written back to memory (see top table in Figure 6). Stalling the pipeline while waiting for the state to be updated would guarantee consistency at the cost of performance (middle table).

To resolve the issue, [58] uses read/modify/write state operations that are performed in a single clock cycle, but at the cost of very constrained update operations. FlowBlaze, on the other hand, leverages the inherent parallelism given by the presence of different flows that access different flow context entries, hence, memory areas. In particular, the scheduler (recall Figure 5) guarantees flow context consistency by conservatively locking the pipeline only when two packets need to access the same flow context. To achieve this, the scheduler recognizes the flow a packet belongs to by using one of the hash keys ($FK1$) generated by the key extractor. When a new packet arrives, the scheduler feeds it to the pipeline if no other packets with the same hash key are being processed. Otherwise, the scheduler stalls the pipeline.

To mitigate any potential head-of-line blocking issues, in our design we arrange the state update ALUs in a parallel fashion. This effectively reduces the number of stalled cycles at the cost of constraining the complexity of state updates. That is, we limit state updates to one single operation per operand, which is anyway sufficient to implement the use cases described in Sec. 3.4. By doing so, as we will see in Sec. 5, a single queue in the scheduling block is enough to achieve the target performance.

Here, notice that this design decision may be changed with relatively minor modifications, e.g., arranging ALUs in a serial fashion, thereby providing richer state update semantics. However, in such a case, the number of stalled cycles would increase, and so would the risk of having head-of-line blocking issues. Thus, our general architecture includes the option of using multiple waiting queues for packets belonging to different flows.

Param.	Value	Descr.
k	4x32b	Flow context's registers
m	8	Maximum number of conditions.
z	64b	Size of metadata moved between elements
h	8x32b	Global registers
ALUs	5	The maximum number of ALUs dictates the maximum number of update functions that can be performed for a given transition.
mqs	20	max queue size, in number of packets
nq	1	number of queues

Table 4: Parameters of a FlowBlaze stateful element in our hardware implementation.

Resource type	Reference switch	FlowBlaze
# Slice LUTs	49436 (11%)	71712 (16%)
# Block RAMs	194 (13%)	393 (26%)

Table 5: NetFPGA's resource requirements for FlowBlaze with a single stateful element compared to those of a reference, single-stage Ethernet switch.

The scheduling block uses $FK1$ to assign packets to Q different queues, guaranteeing that packets belonging to the same flow are always enqueued in the same queue, thus keeping the original ordering for packets belonging to the same flow (cf. Figure 6). The scheduler serves the queues in a round-robin fashion. When a queue is selected, it verifies if a packet with the same hash $FK1$ is already in the pipeline. If that is the case, the scheduler examines the next queue, until it finds a queue whose first packet has a different hash. If no other queues are available, the pipeline is stalled. Otherwise, the scheduler extracts the current queue's first packet and feeds it to the pipeline.

4.4 Hardware Implementation

Our implementation is based on the NetFPGA SUME [65] SmartNIC, an x8 Gen3 PCIe adapter card containing a Xilinx Virtex-7 690T FPGA [3] and four SFP+ transceivers providing four 10G Ethernet links. The system is clocked at 156.25MHz and designed to forward 64B minimum-size packets at line rate. We synthesized FlowBlaze using the standard Xilinx design flow.

Our prototype fixes the machine model's parameters as in Table 4 and uses a non-programmable packet parser, a configurable size flow context table, and a fixed-size EFSM table. The Flow Context Table is implemented with BRAM blocks. Each entry has 128b for the flow key and 146b for the value (16b of state label plus 4 registers of 32b and 2b acting as internal flags). The EFSM table is implemented by a small TCAM of 32 entries of 160 bits. The limited size is due to the technical challenges of implementing a TCAM on FPGAs, which is still an open research issue [36, 59, 37]. Nonetheless, Table 3 shows that such number of entries is al-

ready sufficient for a large range of use cases. As described earlier, the scheduler block has a single queue, for up to 20 packets, which is enough to provide the required throughput and forwarding latency for the tested workloads. We studied the implications of different queue sizes and scheduler configurations in [17].

Table 5 lists the logic and memory resources (in terms of absolute numbers and fraction of available FPGA resources) used by a FlowBlaze implementation with a single stateful element and a Flow Context Table with 16k entries. For reference, we also list those required for the NetFPGA SUME single-stage reference switch, i.e., a simple Ethernet switch. The reported resources include the overhead of several blocks, such as the microcontroller for the FlowBlaze configuration, the input/output FIFO for the 10G interfaces, etc., which are required to operate the FPGA and do not need to be replicated for each element. We successfully implemented on the NetFPGA SUME up to 6 stateful elements for a total of about 200k flow context entries, using around 57% of LUTs and 85% of BRAM blocks.

4.5 Software Implementation

We implemented FlowBlaze's pipeline design as a software data plane to enable any network functions written using the FlowBlaze abstraction to run in software. Briefly, we provide two implementations: a FlowBlaze module on the mSwitch [31] platform in native C (open source [25]); and for the Linux kernel network stack using eBPF/XDP [1]. We evaluate these implementations in the next section.

5 Evaluation

Methodology We experimentally measure FlowBlaze's performance with end-to-end tests and microbenchmarks, resorting to simulations to test corner cases scenarios or to unveil details that would not be visible with black-box testing. To test the workload-dependent behavior of FlowBlaze we used a number of traffic traces collected at various operational networks. Here we report the results for publicly available traces (see Table 6) selected from carrier networks (CHI15 [16], MW15 [46]) and from university datacenters (UNI1, UNI2) [10, 9]. For the hardware implementation tests, the achieved performance is independent of the particular tested application and only influenced by the number of pipeline elements, thus we report tests only for a UDP Stateful Firewall application (NetFPGA-FW-FB). For the software implementations we show the performance of the UDP Stateful Firewall (XDP-FW-FB) and Big Flow Detector applications (mSw.-FB), to compare them with custom implementations of the same functions and evaluate the overhead of the FlowBlaze abstraction. Here, we use as comparison plain mSwitch [31], Linux's XDP [1] and VPP [27] on DPDK. The VPP functions are from the official project repository (VPP-FW), while XDP-FW is a functionally equivalent implementation of the UDP Stateful Firewall function implemented by FlowBlaze.

	Max # active flows			Max # new flows/ms			Mean Pkt
Trace	IP s	IP s,d	5 tpl	IP s	IP s,d	5 tpl	**Size (B)**
UNI1	575	997	4k	13	19	39	697
UNI2	948	3k	7k	20	42	42	751
MW15	12k	130k	152k	38	112	114	540
CHI15	92k	147k	178k	135	144	144	778

Table 6: Max number of active flows for 10s time windows and max number of new flows/ms in the examined traces.

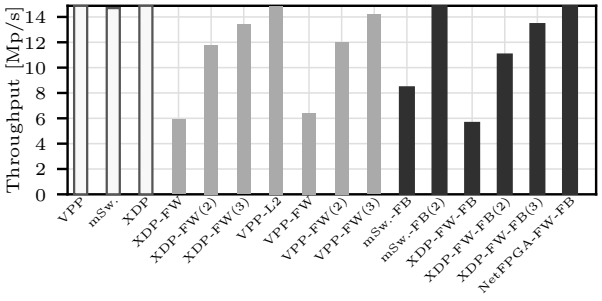

Figure 7: Packet forwarding rates of bare packet I/O engines (white), stateful packet processing w/o FlowBlaze (light gray) and that with FlowBlaze (dark gray): Numbers in () indicate the number of CPU cores if not 1. FlowBlaze does not add overhead (XDP-FW vs XDP-FW-FB) and it scales well (XDP-FW-FB vs XDP-FW-FB(2) and mSw.-FB vs mSw.-FB(2)). The NetFPGA implementation can always forward at line rate while saving up to 3 CPU cores.

Testbed For NetFPGA experiments, we use a single machine equipped with Xeon X3470 CPU clocked at 2.93 Ghz, the quad-port NetFPGA board (cf. Sec. 4) and a single dual-port Intel 82599 10 GbE NIC. Each 10 GbE port is connected to each of two active NetFPGA ports. For experimenting with software implementations, we use two servers connected back-to-back with Ethernet cables: each has an Intel Xeon E3-1231 v3 CPU (3.4GHz) and a single dual-port Intel 82599 10 GbE NIC. One server is used to generate and terminate test traffic; the other is used to forward packets.

5.1 Throughput

End-to-end tests We measure the end-to-end FlowBlaze throughput when running different applications using both our hardware and software implementations. The mSwitch and XDP implementations are configured with the Big Flow Detector and UDP Stateful Firewall, respectively, both identifying flows by the 5-tuple. The same application is configured also on the NetFPGA implementation.

Figure 7 summarizes the measured packet forwarding rates of minimum-sized (64B) packets with various systems. All the *bare* packet I/O frameworks (white bars) achieve line rate (14.88 Mp/s) as they do not touch packet headers. When implementing packet processing logic on top of them, rates decrease (gray bars) and we need more CPU cores to reach line rate. Implementing network functions with FlowBlaze

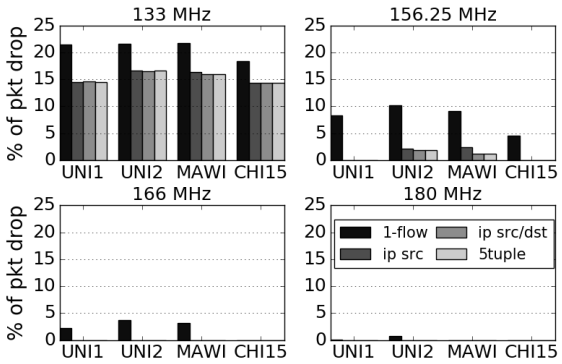

Figure 8: Drop rate of FlowBlaze when clocked at different frequencies and forwarding the traffic traces of Table 6, at 40Gb/s line rate. The 1-flow label shows the case of a pipeline without the scheduler block. With more granular flow definitions, e.g., 5-tuple, there is a higher degree of flow-level parallelism, which reduces the need to stall the pipeline. This can be seen looking at the change of drop rates for different flow definitions in the 156.25MHz case.

(black bars) does not add much overhead, as we see in the comparison between XDP-FW and XDP-FW-FB. Further, FlowBlaze scales to multiple CPU cores well (see XDP-FW-FB and mSw.-FB). Notice that the software implementations are forwarding only 4 flows, and thus we are showing a best case scenario for the achieved forwarding rate. The NetFPGA implementation can forward packets at 10Gbps line rate, and the performance is independent of the number of flows being forwarded. These results show that, even for a relatively simple use case, FlowBlaze can free several CPU's cores from network processing tasks.

Hardware A clock-cycle detailed simulation of the hardware design shows that our prototype could in principle forward 40 Gb/s for all packet sizes when clocked at 156.25MHz. However, the introduction of pipeline stalls to guarantee flow state consistency may actually lower the achieved throughput for some traffic mixes. Thus, in order to have a better understanding of FlowBlaze performance, we use the traces described earlier to test it with different workloads. The traces show a largely bi-modal distribution, with at least 30% (or more) of the packets being minimum size. Furthermore, we try different versions of the design, clocked at 133, 156.25, 166 and 180MHz. This helps to highlight the effect of the packet scheduler, with respect to the option of running the system with a higher frequency.

We use the `moongen` packet generator on a dedicated server to replay the traffic traces, connecting the server's 4 10GbE ports back-to-back with the NetFPGA. The traces are replayed while removing any inter-packet gap, i.e., we generate traffic at 40Gb/s line rate. We consider 4 different flow definitions: 1-flow, i.e., no distinction of network flows and thus no scheduler block, IP src, IP src/dst and 5-tuple. Fig. 8 reports the results.

For a frequency of 133MHz the FlowBlaze pipeline is unable to sustain line rate: the roughly 15% packet drop is independent of the stalling and of flow definitions. Our prototype's selected clock frequency, i.e., 156.25MHz, is the minimum one that sustains 40Gb/s with our design, but stalling reduces the actual throughput: this is visible in the 1-flow case, which results in a 4.6-10.20% packet drop rate, depending on the trace. The introduction of the scheduler block allows FlowBlaze to reduce (or completely remove) packet drops to 0-1.8% when using a 5-tuple flow definition. Slightly rising the frequency to 166MHz allows FlowBlaze to completely remove packet drops for all the traces and all flow definitions. In contrast, in the 1-flow case, even further increasing the frequency to 180MHz does not achieve that, with 0.7% of drops with the UNI2 trace.

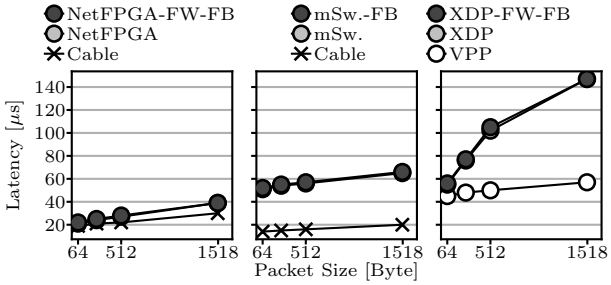

Figure 9: End-to-end RTT: offloading NFs to hardware reduces latency by avoiding PCIe and CPU overheads (dark gray plots). FlowBlaze adds almost no latency to the hardware and software baselines (comparison between light and dark gray plots within each graph).

5.2 Latency

In terms of latency, offloading stateful processing to an I/O peripheral can significantly reduce end-to-end latency by avoiding transfers over PCIe. Recall that end-to-end RTTs between a pair of client and server machines over Ethernet, TCP, a socket API and using a simple HTTP parser are a few tens of µs [32]; RTTs over datacenter fabrics, thanks to sophisticated congestion control algorithms [5, 6, 61] and RDMA deployment [64, 50], can be lower than 100 µs [29].

We connect two 10G NIC regular ports to two NetFPGA ports to measure the end-to-end latency; we then run a netmap `pkt-gen` program on one regular port to send and receive packets while instrumenting another `pkt-gen` on the other regular port to echo back received packets. The measured latency includes two passes through the NetFPGA and the latency of the `pkt-gen` (including overheads of moving packets through the PCIe bus 4 times).

Figure 9 plots the RTT measured at the `pkt-gen` generator. In the left graph the NetFPGA implementation adds only 2–9 µs to cable, and up to 1 µs to plain FPGA forwarding (no FlowBlaze). This is because packets are avoiding PCIe bus transfers and FlowBlaze is optimized to process a packet in just 8 clock cycles. Here, notice that multiple pipelined elements may slightly increase the processing latency of Flow-

Blaze, with each element adding about 50ns. Also, a full queue in the scheduler block may add up to 384ns of waiting time. In any case, even with these additional delays, FlowBlaze packet processing latency is well below one µs.

The middle and right graphs show the RTTs when using mSwitch and XDP implementations. Since software packet processing requires moving packets over the PCIe bus (4 times for round trip), in this case the RTT increases by 38–46 µs with mSwitch (mSw.-FB over Cable) and by 41–127 µs with XDP (XDP-FW-FB over Cable). However, this additional latency does not come from the FlowBlaze abstraction but from packet I/O frameworks; we can see that by comparing them against mSw and XDP, respectively. While not visible in the graphs, we note that FlowBlaze adds up to 1 µs in both the mSwitch and XDP implementations.

In summary, offloading stateful functions to SmartNICs is important for low latency end-to-end services. Furthermore, since FlowBlaze does not add latency to base packet I/O frameworks, operators can start deploying it in software, allowing for incremental deployment of FlowBlaze-enabled SmartNICs that implement performance critical network functions.

5.3 Power Consumption

The NetFPGA consumes 16W when idle and configured with a no-op bitstream. When the FlowBlaze bitstream is loaded, the consumption grows to 22W and is independent of the packet rate and of the network function programmed on FlowBlaze. This consumption has to be considered in addition to the overall system's power consumption, which is 85W when the CPU is idle (For a total of about 107W). In contrast, the software implementations mSwitch and FB-mSwitch consume 124W and 123W of power during operation, respectively, and 119W when not forwarding packets; FB-eBPF consumes 123W in operation and 118W otherwise. In all, FlowBlaze provides significant power savings over software-based implementations, while supporting much higher packet forwarding rates.

5.4 Flow Scalability

The maximum number of entries a FlowBlaze's stateful element can host depends on the amount of state required by the application (e.g., number of registers); our NetFPGA implementation can host about 200k entries, which are enough for all the traces listed in Table 6. It should be noted that using a simpler but less efficient hash scheme for the FlowBlaze design, such as a 4-left hash, would have made the system unable to deal with CHI15. In fact, a 4-left hash table, with 65-70% maximum load factor could only host about 140k entries in the NetFPGA's memory. Further, it is worth noting that the NetFPGA SUME uses a fairly old FPGA generation, with less than 10MB of SRAM blocks. Modern FPGAs could host more than 5x times such number of entries. In any case, related work such as [24] shows that these numbers are in line with current datacenter requirements.

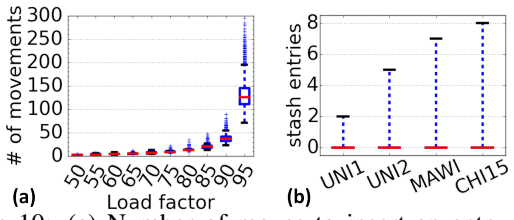

Figure 10: (a) Number of moves to insert an entry in the Flow Context Table. (b) Number of entries in the stash.

5.5 Flow Insertion Performance

Recall that FlowBlaze extends Cuckoo hashing for constant time insertion in hardware. Since slow insertion could lead to dropping flows, we are interested in whether FlowBlaze can handle high flow arrival rates.

To analyze its behaviour, we implement the FlowBlaze insertion algorithm in software, and measure the number of entry movements required for a new entry insertion while increasing occupancy of the Flow Context Table. We run two tests. First, for each traffic trace, we first fill the table to the required occupancy level, e.g., 50%, then we try to insert the next trace's flows and measure the required number of movements for each such insertion. In a second test, we fill the table with randomly generated keys, and then try to insert new keys that are also randomly generated, performing 10k independent measurements. In all cases, the hash table is provided with memory to host all the entries when loaded at 95%. The results are similar for the two tests; for the sake of brevity, Fig. 10a shows them only for the second test.

For a table 95% full, the median number of movements is 125 per insertion, with the highest outlier requiring about 300 movements. Recall that a movement takes 6.4ns, thus 300 movements equates to about 1.9us and so FlowBlaze is able to scale to insertion rates in the order of millions of entries per second. However, recall that FlowBlaze uses a stash that can host at most 8 entries waiting for insertion. If the flow arrival rate is faster than the insertion rate in the hash table, the stash could become full and flows would be dropped. We measured the stash occupancy when using the traffic traces described earlier, simulating a challenging scenario in which each entry insertion in the table takes 450 movements, i.e, 1.5× the worst outlier of Figure 10a. The results in Figure 10b show that in all the cases, with the most fine-grained 5-tuple flow definition, the stash has at most 8 entries (meaning no flow is dropped) while being empty for most of the time.

6 Discussion

Ease of use While we did not run large scale surveys, we can report our experience in using FlowBlaze to implement network functions. In line with the findings of [40], we found the EFSM abstraction requires the programmer to adapt to a model that is different from regular procedural programming, but relatively simple to adopt. Once adopted, the EFSM model helps in focusing on the required function's

state and on how it evolves. We found this particularly helpful in designing more complex functions, such as the one of Fig. 4. Here, we would first focus on the different states the function could be in, and then describe the inputs that would make a function evolve from one state to the next, leading to a very linear design and implementation process.

Security Even if a smart state encoding, such as the one adopted in [49], could enable handling state for millions of flows, FlowBlaze will always have some hard memory limit which could be potentially exploited, for example by DoS attacks. To deal with this, FlowBlaze provides primitives that allow a programmer to explicitly handle cases where the Flow Context Table is full. Still, it is up to the programmer to define functions that are robust to DoS attacks. For example, the function in Figure 4 could be preceded by an element that implements a SYN flooding detection function so that traffic from a host that performs a SYN attack could be dropped, avoiding the creation of a large number of entries in the Flow Context Tables for the elements that follow.

7 Related Work

The MAT abstraction is the starting point for FlowBlaze and was formalized by OpenFlow [48]. RMT [14] extends that model with programmable packet parsers, re-configurable forwarding tables to match on different headers and programmable actions. dRMT [18] is an alternative implementation of the RMT instruction set for a high-performance switching ASIC. Languages such as P4 [13] and PX [15] are used to describe such configurable MATs. Our work extends RMT to perform stateful packet processing.

SNAP [7] introduces a stateful packet processing abstraction for the control plane. The network is programmed as if it was a single big stateful switch, then, a compiler distributes state variables to the network devices. State is represented by an array of values that can be indexed by e.g., packet header fields' values. SNAP does not focus on the implementation of stateful operations in the data plane, which is instead the focus of our work. In fact, FlowBlaze can be used as an implementation target for SNAP. However, FlowBlaze provides a predefined per-flow state access model, which may not suite all the SNAP's programs. That is, SNAP's abstraction is less constrained, therefore there may be programs that cannot be mapped to a FlowBlaze target.

Perhaps the closest works to FlowBlaze are FAST [51] and OpenState [11]. Both works define an FSM abstraction, but (i) do not define a state access model that allows for both per-flow and global consistency, and (ii) do not deal with issues related to the integration of FSMs in an RMT machine model. FlowBlaze fills these gaps and provides a hardware implementation that addresses problems that have to do with quick per-flow state insertion and state update consistency. In contrast, FAST provides only a software implementation, while OpenState can only support much simpler Mealy Machines.

VFP [23] presents a MAT-like abstraction, GFT, to offload some network functions to a SmartNIC, and defines concepts similar to the *biflow* we use in FlowBlaze. AccelNet[24] implements GFT in a FPGA-based SmartNIC. Their design differs from FlowBlaze in several ways. First, AccelNet requires the first packet of a flow to be handled in software. For short flows this introduces additional delays that may be critical for real-time applications, thus FlowBlaze allows for functions to be entirely executed in the FPGA instead. Second, AccelNet uses a small (2k flows) cache in SRAM backed by a larger DRAM to host flow entries. This is also related to Marple, which extends data plane state memory using off-chip DRAM [53] to support network monitoring functions in high-performance switching ASICs. In contrast, FlowBlaze places all the flow entries in a highly optimized hash-table in SRAM, which guarantees constant delays for all flows' state reads and writes. Unlike FlowBlaze, AccelNet does not provide an abstraction to program FPGA functions, though it does implement a microcode programmable action block which allows for reconfiguration of the actions without requiring a change in the FPGA design; this is complementary to our work.

Examples of functions offloaded to programmable hardware are presented in [49, 57, 43, 21, 44, 56, 38]. FlowBlaze provides an abstraction to implement such use cases, and addresses issues in those implementations to do with control plane scalability and limited state memory. We described the implementation of a few datacenter use cases with an earlier software version of FlowBlaze in [12].

8 Conclusion

We presented FlowBlaze, an EFSM-based abstraction able to describe network functions targeted at high-performance data plane implementations. FlowBlaze is flexible and can implement complex network functions while being compatible with the wide-spread MATs pipeline abstraction. Leveraging the flow state concept, we provided an efficient hardware implementation that can run, at line-rate, stateful network functions that keep large, per-flow state. FlowBlaze is built on top of the NetFPGA open platform and both hardware and software sources are publicly available [25].

Acknowledgements

We would like to thank the anonymous NSDI reviewers and our shepherd Anirudh Sivaraman for their valuable feedback. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 761493 ("5GTANGO") and No. 762057 ("5G-PICTURE"). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt.

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

[3] Virtex-7 Family Overview. http://www.xilinx.com.

[4] V. S. Alagar and K. Periyasamy. *Extended Finite State Machine*, pages 105–128. Springer London, London, 2011.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[6] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.

[7] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[8] AT&T, BT, CenturyLink, China Mobile, Colt, Deutsche Telekom, KDDI, NTT, Orange, Telefom Italia, Telefonica, Telstra, and Verizon. Network function virtualization - white paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf.

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

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM IMC*, ACM SIGCOMM IMC '10, 2010.

[11] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM CCR*, 44(2):44–51, 4 2014.

[12] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi. Implementing advanced network functions with stateful programmable data planes. In *Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on*, pages 1–2. IEEE, 2017.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[14] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM '13*, ACM SIGCOMM '13, pages 99–110. ACM, 2013.

[15] G. Brebner and W. Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, Jan 2014.

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

[17] C. Cascone, R. Bifulco, S. Pontarelli, and A. Capone. Relaxing state-access constraints in stateful programmable data planes. *ACM SIGCOMM Computer Communication Review*, 48(1):3–9, 2018.

[18] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

[19] T. P. L. Consortium. The p4 language specification - version 1.0.5, 5 2018.

[20] T. P. L. Consortium. P416 language specification, 5 2018.

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

[22] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, January 2006.

[23] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[25] FlowBlaze. Repository with FlowBlaze source code and additional material. `http://axbryd.com/FlowBlaze.html`.

[26] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, Feb 2005.

[27] L. Foundation. FD.io. `https://fd.io/`.

[28] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE ANCS '13*.

[29] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[30] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, volume 2001, 2001.

[31] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mswitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ACM SOSR '15, pages 1:1–1:13. ACM, 2015.

[32] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *Proc. USENIX NSDI*, 2018.

[33] H. Hong, H. Choi, D. Kim, H. Kim, B. Hong, J. Noh, and Y. Kim. When cellular networks met ipv6: Security problems of middleboxes in ipv6 cellular networks. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 595–609, April 2017.

[34] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[35] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, 2017. USENIX Association.

[36] B. Jean-Louis. Using block RAM for high performance read/write TCAMs, 2012. Xilinx XAPP204.

[37] W. Jiang. Scalable ternary content addressable memory implementation using fpgas. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 71–82, Oct 2013.

[38] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[39] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[40] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, 2015. USENIX Association.

[41] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, Dec. 2009.

[42] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *ACM SIGCOMM '16*.

[43] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, 2016. USENIX Association.

[44] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev.*, 51(2):795–809, Apr. 2017.

[45] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems*

Design and Implementation, USENIX NSDI'14, pages 459–473. USENIX Association, 2014.

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

[47] N. McKeown. Programmable forwarding planes are here to stay. In *ACM SIGCOMM 2017 The Third Workshop on Networking and Programming Languages (NetPL)*, 2017.

[48] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 3 2008.

[49] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[50] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.

[51] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ACM HotSDN '14, pages 61–66. ACM, 2014.

[52] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.

[53] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM.

[54] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, 2017. USENIX Association.

[55] Z. Qian and Z. M. Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 347–361, Washington, DC, USA, 2012. IEEE Computer Society.

[56] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[57] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, 2017. USENIX Association.

[58] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.

[59] Z. Ullah, M. Jaiswal, Y. Chan, and R. Cheung. FPGA Implementation of SRAM-based Ternary Content Addressable Memory. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE IPDPSW 2012, 2012.

[60] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[61] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[62] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 34:1–34:13, New York, NY, USA, 2015. ACM.

[63] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ACM CoNEXT '13, pages 97–108. ACM, 2013.

[64] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and

M. Zhang. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 523–536. ACM, 2015.

[65] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro '14*, 34(5):32–41, 2014.

A: ALU Instructions

Type	Instructions	Description
Logic	NOP	do nothing
	NOT	$OUT \leftarrow NOT(IN1)$
	XOR, AND, OR	$OUT \leftarrow IN1 op IN2$
Arithmetic	ADD, SUB	$OUT \leftarrow IN1 op IN2$
	ADDI, SUBI	$OUT \leftarrow IN1 op IMM$
Shift/Rotate	LSL (Logical shift left)	$OUT \leftarrow IN1 << IMM$
	LSR (Logical shift right)	$OUT \leftarrow IN1 >> IMM$
	ROR (Rotate right)	$OUT \leftarrow IN1 ror IMM$

Table 7: FlowBlaze ALU-supported instructions.

B: Pipeline Simulations

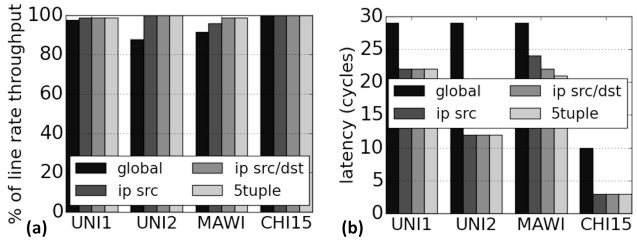

Figure 11: *sim1* (a) throughput and (b) latency.

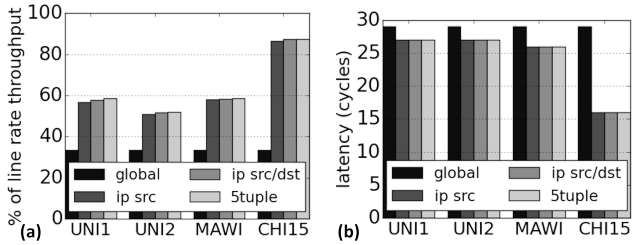

Figure 12: *sim2* (a) throughput and (b) latency.

In order to understand FlowBlaze's design performance (when clocked at 156.25MHz) with respect to a worst case scenario, we implemented a custom FlowBlaze simulator and used modified versions of the traces described in Sec. 5. The traces show a largely bi-modal distribution, with at least 30% (or more) of the packets being minimum size. It should be noted that we used this custom pipeline simulator since a clock-cycle level hardware simulator would be too slow to perform trace-based simulations at this scale.

We perform two simulations: *sim1* simulates line rate by removing any time gap between packets; this results are used to validate the simulator and are comparable to our experimental evaluation results (cf. Fig. 8); *sim2* modifies all packets to be minimum size and so represents a worst-case workload. Figures 11 and 12 plot throughput relative to FlowBlaze's line rate, with the scheduler block (cf. Section 4.3) and without it (global label), as well as the 99^{th} percentile forwarding latency which is increased by pipeline stalls or queuing. The former case is tested against three flow definitions: IPsrc, IPsrc-IPdst, and 5-tuple. In the global label case, we force the pipeline to stall for 3 cycles for every minimum size packet in order to guarantee state consistency (recall the middle table in Figure 6).

The results show the effects of having the scheduler block (cf. traces details in Table 6): depending on flow definitions and traces, it improves the throughput by 50–160% in the worst case (*sim2*), and up to 12% in *sim1*. In this last case, the scheduler reduces the per-packet latency by 30–70% .