HyPaFilter+: Enhanced Hybrid Packet Filtering Using Hardware Assisted Classification and Header Space Analysis

Andreas Fiessler, Claas Lorenz, Sven Hager, Björn Scheuermann, and Andrew W. Moore, *Member, IEEE*

***Abstract*— Firewalls, key components for secured network infrastructures, are faced with two different kinds of challenges: first, they must be fast enough to classify network packets at line speed, and second, their packet processing capabilities should be versatile in order to support complex filtering policies. Unfortunately, most existing classification systems do not qualify equally well for both requirements: systems built on special-purpose hardware are fast, but limited in their filtering functionality. In contrast, software filters provide powerful matching semantics, but struggle to meet line speed. This motivates the combination of parallel, yet complexity-limited specialized circuitry with a slower, but versatile software firewall. The key challenge in such a design arises from the dependencies between classification rules due to their relative priorities within the rule set: complex rules requiring software-based processing may be interleaved at arbitrary positions between those where hardware processing is feasible. Therefore, we discuss approaches for partitioning and transforming rule sets for hybrid packet processing. As a result, we propose HyPaFilter+, a hybrid classification system consisting of an FPGA-based hardware matcher and a Linux `netfilter` firewall, which provides a simple, yet effective hardware/software packet shunting algorithm. Our evaluation shows up to 30-fold throughput gains over software packet processing.**

***Index Terms*— Packet classification, FPGA hardware accelerator, firewall, header space analysis.**

I. Introduction

SOFTWARE firewalls like `netfilter/iptables` [1], `pf` [2], or `ipfw` [3] are widely used in practice, both in stand-alone applications and as the basis for professional security appliances [4]. Their main advantages are flexibility and powerful filtering options, as well as their easy setup and handling, since they can be used on top of common operating systems with low-cost, commercial off-the-shelf (COTS) hardware. These CPU-based architectures, however, hardly meet the line rate packet processing requirements for high link speeds such as 40 Gbit/s and beyond, which leave only small processing time frames of 8 ns or less for each packet in the worst case [5]. In contrast, packet classification systems based on special purpose hardware, such as network processors (NPUs) [6], [7], field-programmable gate arrays (FPGAs) [5], [8]–[10], graphics processing units (GPUs) [11], or application-specific integrated circuits (ASICs) [12] provide an abundant amount of parallelism which can be used to process many network packets at once. Furthermore, the matching process for every single packet is often parallelized, leading to throughput rates of up to 640 Gbit/s [12].

However, dedicated hardware is significantly more constrained with respect to the expressiveness of the supported rule set semantics: while the functionality of software-based classification systems ranges from stateful connection tracking over probabilistic matching to deep packet inspection [1]–[3], specialized hardware engines are often restricted to simple stateless packet classification with no or only limited connection tracking capabilities [5], [9]–[12]. Moreover, while software firewalls can utilize large amounts of memory for storing policies and connection states, hardware firewalls have to operate within fixed boundaries.

In order to combine the advantages of massively parallel matching hardware and the powerful inspection capabilities of software-based packet filters, we propose *HyPaFilter+*, a hybrid packet classification concept. The HyPaFilter+ approach aims to reach the packet rate and processing latency of a dedicated hardware firewall for common, easy to classify traffic, while providing the flexibility and functionality of a software firewall for packets that require complex processing. The hardware part of the hybrid system processes all traffic first and is able to *shunt* packets, i.e., hand them over to the software when necessary. To this end, HyPaFilter+ partitions a user-defined packet processing policy into a part consisting of *simple rules*, manageable by specialized matching hardware, and a part consisting of *complex rules*, which requires handling in software. We found that a key challenge in such a hybrid design, regardless of its concrete implementation, is the proper handling of dependencies between different rules in the specified policy: if the hardware detects a rule match of an incoming packet in the simple part of the policy, it must ensure that the packet does not match a more highly prioritized rule installed in the software filter *before*

Manuscript received August 29, 2016; revised March 24, 2017 and July 30, 2017; accepted September 3, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Bremler-Barr. Date of publication September 27, 2017; date of current version December 15, 2017. This work was supported in part by the German Federal Ministry for Economic Affairs and Energy, in part by the German Federal Ministry of Education and Research, and in part by the EU Horizon 2020 SSICLOPS Project under Grant 644866. *(Corresponding author: Andreas Fiessler.)*

A. Fiessler and C. Lorenz are with genua GmbH, 85551 Kirchheim, Germany (e-mail: andreas_fiessler@genua.de; claas_lorenz@genua.de).

S. Hager and B. Scheuermann are with the Department of Computer Engineering, Humboldt University of Berlin, 12489 Berlin, Germany (e-mail: hagersve@informatik.hu-berlin.de; scheuermann@informatik.hu-berlin.de).

A. W. Moore is with the Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, U.K. (e-mail: andrew.moore@cl.cam.ac.uk)

Digital Object Identifier 10.1109/TNET.2017.2749699

the action specified by the hardware-detected rule is applied. However, it is desirable to avoid a full-fledged software packet classification whenever possible in order to achieve the full hardware speedup for a large number of packets. To overcome this challenge, the HyPaFilter+ approach determines for which packets a hardware-only classification is safely possible. Furthermore, even if software processing for a packet is required, the matching information from the hardware can be reused in order to narrow down the set of rules the software filter has to match against this packet. We also address policy updates, as both the simple and complex part of a policy can change at arbitrary positions after an initial system setup.

The achievable performance of HyPaFilter+ depends on both the structure of the implemented policy and the network traffic characteristics. However, previous examination of real-world traffic in [13] showed that the fraction of traffic which can be analyzed by simple packet filter rules is large enough to expect a significant performance gain in practical applications. Our evaluation results indicate that the HyPaFilter+ system, which we prototyped using a NetFPGA SUME board [14] and a Linux host system, can increase the maximum achievable classification throughput over a software-only approach by a factor of up to 30. This holds even for policies with many and widely spread complex rules. In summary, the main contributions of this work are:

- We present a hybrid packet classification concept which combines the benefits of dedicated matching hardware with powerful matching semantics typically found in software-only approaches.
- We describe an effective dispatch technique based on formal rule set analysis that enables hardware-only processing for large traffic fractions. That way, we significantly reduce the number of expensive FPGA-to-host communications.
- To mitigate the processing costs for packets that must be shunted to the host, we leverage a hardware-assisted binary search in order to narrow down the number of rules which have to be checked in software.

HyPaFilter+ extends our previously described HyPaFilter architecture [15] by adding a packet dispatch algorithm which uses a form of *Header Space Analysis* [16] on the rule set to reduce expensive FPGA-to-host communication.

The remainder of this paper is structured as follows: in Section II, we discuss related work. Next, we briefly introduce the packet classification problem in Section III. Section IV describes the hardware matching circuitry, Section V shows how to optimize the internal shunting decision, and Section VI illustrates approaches to optimize the software packet matching process for shunted packets. Section VII describes the overall architecture of the HyPaFilter+ system. Finally, we present our evaluation results in Section VIII and conclude this paper in Section IX.

II. Related Work

Network packet classification has been of major interest to the research community due to its importance for packet-switched networks [17], [18]. Most scientific work in this area focuses on the geometric variant of the packet classification problem, which considers a limited number of packet header fields and does not take other criteria into account, such as packet payloads or connection states. These works can be roughly split into the following categories: *classification algorithms*, *hardware architectures*, and *rule set transformations*.

Classification algorithms traverse an algorithm-specific data structure in order to find the most highly prioritized rule that matches on all relevant header fields of an incoming packet. Such approaches exist in many different flavors, such as decision tree algorithms [19], [20], bit vector searches [21], [22], or techniques based on hash maps [23]. In comparison to a straightforward linear search through the rule set, these advanced classification algorithms provide significantly faster classification performances [17]. Despite this fact, many practically used packet classification systems, such as `netfilter` [1] and `pf` [2], implement a linear search in order to discriminate network packets and thus generally suffer from low classification performance [24]. However, these systems also provide powerful rule set semantics which are more expressive than plain stateless header field inspection.

Specialized hardware architectures used for packet classification typically employ large amounts of parallelism in order to achieve high throughput rates. The most common hardware architecture used for packet classification is *Ternary Content Addressable Memory (TCAM)*, which matches the entire rule set in parallel against incoming packet headers [25] and can thus process every incoming packet in a small, fixed number of clock cycles. On the downside, TCAMs are expensive, power-intensive, and cannot natively represent rules with range or negation tests [10]. Other widely used implementation platforms for packet classification are FPGAs [5], [8]–[10], NPUs [6], [7], and GPUs [11], which typically also employ a full parallel match [10] or implement a parallelizable classification algorithm [5]–[9], [11]. Although significantly faster than software-based systems, these approaches only support limited, stateless matching semantics. In contrast, the HyPaFilter+ design combines the flexibility of existing software engines with the processing speed of dedicated hardware.

Rule set transformation techniques are orthogonal to the employed classification algorithm/architecture. The goal is to transform an initial rule set $\mathcal{R}$ into an equivalent rule set $\mathcal{R}'$ that can be traversed faster for incoming network packets. Existing approaches for rule set transformation are rule set minimization [26] or the encoding of decision tree data structures into the rule set [24]. HyPaFilter [15] and HyPaFilter+ utilize the latter transformation variant to install complex rules in the software filter that can reuse the hardware classification result to accelerate the software matching.

The possibility of hybrid packet filters for FPGA/`netfilter` and NPU/`netfilter` combinations has been previously addressed in [13] and [27], respectively. However, these works do not answer the following key questions: (1) How should a packet processing policy be deployed in a hybrid system in order to achieve high classification performance? (2) How does the hybrid system implement rule set updates? The HyPaFilter and HyPaFilter+ approaches answer these questions by providing rule set partitioning schemes which

install simple rules on the FPGA. Complex rules are installed in the software filter on the host system. The complex rules are structured in a way that partial matching information, which has been previously computed by the FPGA, can be reused in the software filter. Moreover, HyPaFilter+ extends HyPaFilter by statically analyzing the rules installed on the FPGA to minimize the number of packets that must be processed in software.

In [19] and [21], geometric rule set representations are used as the foundation of fast search data structures for packet classification. The authors of [16] propose a methodology to quickly identify network malfunctions, such as forwarding loops, which is also based on the geometric model. The static rule set analysis performed by HyPaFilter+ is inspired by the geometric representation used in these works, but applied in a different way: by analyzing geometric rule interdependencies, we significantly narrow down the number of packets which must be shunted at runtime.

III. Problem Statement

In this section, we first introduce the packet classification problem, which serves as the vantage point for the extended packet classification problem, which we define subsequently.

A. Packet Classification Problem

The packet classification problem, as it is most often seen in the literature [10], [19], [22], [23], can be formally defined as follows: let $\mathcal{H} = (H_1 \in D_1, \ldots, H_K \in D_K)$ be a tuple of *header values* and $\mathcal{R} = \langle R_1, \ldots, R_N \rangle$ be an ordered list of *rules* R_i, which is called the *rule set*. Here, D_j is called the *domain* of the jth header field, and $\mathcal{U} = D_1 \times \ldots \times D_K$ is the set of all possible packet headers. For the remainder of this paper, we assume that each D_j is a range of non-negative integers, in order to cover common header fields like IP addresses, ports, or protocol numbers. Every rule R_i consists of K *checks* $C_i^j : D_j \rightarrow \{\text{true}, \text{false}\}$ with $R_i = C_i^1 \wedge \ldots \wedge C_i^K$. R_i is said to *match* the header tuple $\mathcal{H}$ (which we denote by $R_i(\mathcal{H})$) if $C_i^j(H_j)$ yields true for all $j \in \{1, \ldots, K\}$. The goal of the packet classification problem is to find the smallest index i^* such that rule R_{i^*} matches $\mathcal{H}$. This index can subsequently be used in order to look up and execute an *action* for the corresponding matching rule, such as DROP or ACCEPT. Here, the checks C_i^j are assumed to be equality, range, or subnet tests, which are the most common types of tests used in rule sets [21], [24]. Every check C_i^j can be represented as an interval test $H_j \in \left[X_i^j, Y_i^j\right]$, with $X_i^j \leq Y_i^j$ and $X_i^j, Y_i^j \in D_j$. Accordingly, every rule R_i has a geometric representation $G(R_i) \subseteq \mathcal{U}$, which is the K-dimensional hypercube $\left[X_i^1, Y_i^1\right] \times \ldots \times \left[X_i^K, Y_i^K\right]$. We say that two rules R_i and R_k $(i \neq k)$ *conflict* iff there is a header tuple $\mathcal{H}$ that matches both rules, i.e., iff $\exists \mathcal{H} \in \mathcal{U} : R_i(\mathcal{H}) \wedge R_k(\mathcal{H})$, or equivalently, $G(R_i) \cap G(R_k) \neq \emptyset$.

B. Extended Packet Classification Problem

Practical packet filter implementations, such as `netfilter` [1], `pf` [2], and `ipfw` [3], support advanced matching criteria in order to increase the expressiveness of an implemented filtering policy. Examples for such sophisticated checks are connection tracking, rate limiting, unicast reverse path forwarding (URPF) verification, probability-based matching, or deep packet inspection. When used in conjunction with the previously defined basic checks, these tests can greatly foster both the filtering granularity and effectiveness of the used packet filtering policy. In such a system, a rule R_i can be modelled as $R_i = C_i^1 \wedge \ldots \wedge C_i^K \wedge K_i$, where K_i is a rule-specific combination of advanced matching criteria. Thus, each rule R_i consists of simple checks C_i^j, as defined in Section III-A, and a (potentially empty) collection of arbitrarily shaped advanced checks K_i. For each rule R_i we define the *geometric reduction* R_i^-, which projects R_i to its simple checks C_i^j and removes the advanced part K_i. We call a rule R_i a *simple rule* iff $R_i = R_i^-$, i.e., if it does not define advanced checks, otherwise we call R_i a *complex rule*. For a given rule set $\mathcal{R}$, we denote the sublist of simple rules by $\mathcal{R}_S$, and the sublist of complex rules by $\mathcal{R}_C$. Also, for every simple rule $R_i \in \mathcal{R}$, let $\mathcal{R}_S(i)$ be the index of rule R_i in $\mathcal{R}_S$.

```
-d 1.2.3.4/32 -p tcp --dport 80       -j ACCEPT   # Rule R1
#-------------------------------------------------------------
-s 1.2.3.0/24 -p tcp                  -j ACCEPT   # Rule R2
#-------------------------------------------------------------
-d 1.2.3.5/32 -p udp --dport 3306 -m string       # Rule R3
   --string "SELECT" --algo bm        -j DROP     #
#-------------------------------------------------------------
-d 1.2.3.4/32 -p tcp --dport 443      -j ACCEPT   # Rule R4
#-------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53 -m string         # Rule R5
   --hex-string "|11|2|00|" --algo bm -j DROP     #
#-------------------------------------------------------------
-d 1.2.3.6/32 -p udp --dport 53       -j ACCEPT   # Rule R6
```

Listing 1. Example rule set $\mathcal{R}$ in `iptables` syntax.

Listing 1 shows an example rule set $\mathcal{R} = \langle R_1, R_2, R_3, R_4, R_5, R_6 \rangle$ that consists of the simple sublist $\mathcal{R}_S = \langle R_1, R_2, R_4, R_6 \rangle$ and the complex sublist $\mathcal{R}_C = \langle R_3, R_5 \rangle$. Each rule specifies simple checks on source or destination addresses, the transport layer protocol, or the destination port (indicated through the flags `-s`, `-d`, `-p`, or `--dport`, respectively). Rules R_3 and R_5 are complex, as they define string matches on the packets' payload. It can be seen that the simple rules R_4 and R_6 are located at the indices 3 and 4 in $\mathcal{R}_S$, hence $\mathcal{R}_S(4) = 3$ and $\mathcal{R}_S(6) = 4$.

IV. Hardware Filter

In order to support good classification performance, short rule set update latencies, and expressive rule set semantics, the HyPaFilter+ system relies on a hybrid matching algorithm that first processes every incoming packet on the FPGA. After the packet is matched, the FPGA circuitry decides whether the packet requires further, potentially more complex processing in the host-based `netfilter` system.

The classification system implemented on the FPGA solves the packet classification problem on the simple rule set $\mathcal{R}_S$, as introduced in Section III-A. It therefore implements every simple rule $R_i \in \mathcal{R}$. In order to achieve high matching performance on the FPGA with a low, deterministic processing

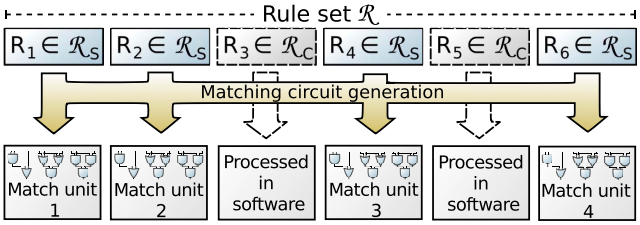

Fig. 1. Translating simple rules of Listing 1 into match units.

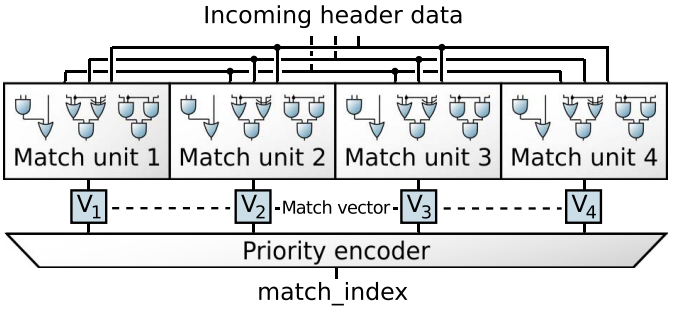

Fig. 2. Parallel match of packet header data against $\mathcal{R}_S$.

latency per packet, we decided to use a rule-set-specific parallel matching engine, which is generated by translating every simple rule $R_i \in \mathcal{R}_S$ at setup time into a specialized match unit $M_{\mathcal{R}_S(i)}$ specified in VHDL, similar to the technique proposed in [10]. Recall that $\mathcal{R}_S(i)$ is the index of rule R_i within the sublist $\mathcal{R}_S$. This process is illustrated in Figure 1. Since each rule in $\mathcal{R}_S$ is a conjunction of simple checks, such as subnet tests or port range tests, the match units are composed of a small number of basic comparator circuits. For example, a rule which matches TCP packets if the source IP address is in the subnet $203.0.0.0/8$ with destination port 80 is translated into three specific comparator circuits: the first one compares the packet's transport protocol field against the TCP transport protocol number 6, while the second and third comparators compare the first octet of the packet's source IP address against 203 and the packet's destination port against 80, respectively. Finally, the results of these comparators are ANDed to determine whether the rule matches.

As the match units are arranged in parallel, incoming network packets can be matched against the entire simple rule set $\mathcal{R}_S$ in a single clock cycle, which yields a result bit vector V_{res} of size $|\mathcal{R}_S|$. Here, the entry at position $\mathcal{R}_S(i)$ of the result vector V_{res}, which we denote by $V_{\text{res}}[\mathcal{R}_S(i)]$, stores a 1 if rule R_i matches the current packet, and a 0 otherwise. As we are interested in the most highly prioritized matching rule, we employ a priority encoder to determine the index of the first enabled bit in V_{res}, which we will refer to as *match_index*. The hardware matching process is sketched in Figure 2.

We opted to use the above described rule-set-specialized matching circuitry in our FPGA-based prototype system due to its small hardware resource footprint. In comparison to generic hardware matching techniques with comparable throughput, such as StrideBV [28] or TCAMs [25], tailormade matching circuitry is significantly smaller and dissipates less power when implemented on an FPGA [29], [30]. During our experiments, the design toolchain was not able to generate a native TCAM implementation on the FPGA for capacities as low as 100 IPv6-capable rules without timing errors, while the tailormade matcher could support several thousands of rules. Nevertheless, we point out that it is of course possible to step away from an FPGA implementation platform and instead use a different hardware matcher for simple rules, e. g., on the basis of a high-density ASIC-based TCAM.

V. Packet Shunting

Up to this point, the packet classification problem is solved for the simple rule set $\mathcal{R}_S$ solely in hardware, as the *match_index* can be used in order to quickly look up the action $A_{\mathcal{R}_S,P}$ that must be applied for the current packet P. If the installed rule set $\mathcal{R}$ does not specify any rules with complex checks, i.e., if $\mathcal{R}_C = \emptyset$ and thus $\mathcal{R}_S = \mathcal{R}$, then the classification is complete at this point and $A_{\mathcal{R}_S,P}$ is applied to the current packet. However, if $\mathcal{R}_C \neq \emptyset$, then additional processing may be required on the host system. Accordingly, some packets must be *shunted* from the FPGA device to the software filter to compute the correct classification result. However, as software-based classification of shunted packets is expensive (in comparison to FPGA-only packet processing), the number of shunted packets should be as small as possible. Furthermore, the shunting decision itself should be computed at line speed in order not to bottleneck the packet pipeline. Thus, in this section, we present two line-speed shunting techniques: *index-based shunting* and *selective shunting*.

A. Index-Based Shunting

The index-based shunting technique, which was originally introduced in our previous work [15], partitions the simple rule set $\mathcal{R}_S$ into an *unambiguous* rule set prefix and an *ambiguous* rule set suffix. When the FPGA detects that an incoming packet P matches a rule in the unambiguous part, it is processed entirely in hardware. Otherwise, P is shunted to the host system for further processing. Here, a simple rule $R_i \in \mathcal{R}$ is called *ambiguous* if there exists a complex rule $R_j \in \mathcal{R}$ with $j < i$, otherwise R_i is called an *unambiguous* rule. Thus, a packet P is shunted to the software classification system whenever a complex rule in $\mathcal{R}_C$ installed on the host system *could* match the current packet P with a higher rule priority than the matching rule in $\mathcal{R}_S$. In the following, we will denote the smallest index of a rule in $\mathcal{R}$ with complex checks by the term *shunt_index*.

In the example shown in Figure 1, the unambiguous prefix consists of the rules R_1 and R_2. In contrast, the ambiguous suffix contains the rules R_4 and R_6, as the complex rule R_3 is more highly prioritized and could potentially conflict with R_4 or R_6, respectively. For instance, consider the case that the hardware matching circuit for the rule set sketched in Figure 1 computes that *match_index* is 3 for an incoming packet P (that is, the packet matches the simple rule R_4). In this case, our hardware classification might be incorrect, as the complex rule R_3 could also match on the packet P. Thus, whenever *match_index* $\geq$ *shunt_index*, index-based shunting sends the classified packet to the host for further processing.

This technique computes the correct classification result in every case, since packets that might match a complex rule

are always shunted to the host system. Also, changes in the complex part of the rule set only require to update the register on the FPGA that holds the *shunt_index*. A major drawback of this technique comes into effect if complex rules appear with a high priority, which forces the shunting of all packets that match only lower prioritized rules.

B. Selective Shunting

Index-based shunting, as introduced in the previous section, can lead to situations where packets are shunted to the host system although they cannot match more highly prioritized complex rules. For example, a TCP packet that matches on rule R_4 in Listing 1 cannot match the complex rule R_3, since R_3 and R_4 are mutually exclusive. Nevertheless, using index-based shunting, the packet would still be shunted to the host because R_3 is more highly prioritized than R_4. This, in turn, leads to a higher workload on the software classifier and can eventually result in throughput penalties.

In this section we introduce *selective shunting*, a methodology to optimize the shunting decisions taken on the FPGA based on a formal *header space analysis (HSA)*. The selective shunting technique leverages the geometric representations (i.e., the header spaces) of rules to narrow down the number of packets that must be shunted to the host. To this end, we compute a *shunting vector* $V_{\mathcal{R}_S}$ that stores a single *shunt bit* for every simple rule R_j in $\mathcal{R}_S$. In the following, we denote the jth bit in $V_{\mathcal{R}_S}$ by $V_{\mathcal{R}_S}[j]$. Also, we denote the set of complex rules in $\mathcal{R}$ that are more highly prioritized than R_j by Γ_j. A set shunt bit $V_{\mathcal{R}_S}[j]$ indicates that there exists at least one possible packet header that matches both the simple rule R_j and at least one complex rule in Γ_j. Hence, R_j conflicts with at least one rule in Γ_j. Using the geometric representations of rules, the layout of $V_{\mathcal{R}_S}$ can be expressed by

$$V_{\mathcal{R}_S}[j] = \begin{cases} 1 & \text{if } \bigcup\limits_{R \in \Gamma_j} \left(G(R^-) \cap G(R_j)\right) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

After the shunting vector $V_{\mathcal{R}_S}$ has been computed in a preprocessing step, it is stored on the FPGA. Each time a packet is classified by the FPGA matching circuitry, we use the determined *match_index* in order to look up the shunt bit $V_{\mathcal{R}_S}[\textit{match\_index}]$. This is in contrast to index-based shunting, where we compared *match_index* with *shunt_index*. Only if the shunt bit is set, the packet is sent to the host for further classification, because the matching simple rule could be overruled by a more highly prioritized complex rule. Otherwise, it can be safely treated entirely in hardware.

We use Figure 3 to visualize the difference between index-based and selective shunting in a two-dimensional header space example. With index-based shunting, all packets are shunted due to the complex rule R_1 with the highest priority. In contrast, with selective shunting only those packets are shunted that match the simple rule R_5 on the FPGA, since R_5 is the only simple rule whose geometric shape intersects with those of the complex rules R_1 or R_3.

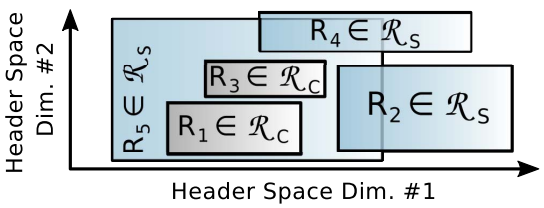

Fig. 3. Example sketch for different rules in a reduced two-dimensional header space.

Algorithm 1 Compute the Shunt Vector From the Rule Set $\mathcal{R}$

```
1: function SELECTIVE_SHUNTING_ANALYSIS(Rule set R)
2:     V_RS ← [ ]    // Initialize V_RS with an empty vector
3:     for i ∈ {1, ..., |R|} do
4:         if IS_SIMPLE_RULE(R[i]) then
5:             bit ← 0
6:             for j ∈ {1, ..., i − 1} do
7:                 if R[j]⁻ ≠ R[j] then // check if R[j] is complex
8:                     if G(R[j]⁻) ∩ R[i] ≠ ∅ then
9:                         bit ← 1
10:            V_RS ← V_RS + [bit]    // Append bit to V_RS
11:    return V_RS
```

The procedure SELECTIVE_SHUNTING_ANALYSIS (SSA) for the computation of $V_{\mathcal{R}_S}$ is shown in Algorithm 1. It can be seen that SSA appends one bit to the shunting vector $V_{\mathcal{R}_S}$ for every simple rule in the input rule set $\mathcal{R}$. For each simple rule, the bit is computed by testing whether the intersection of the geometric representation of the simple rule with the geometric representation of any more highly prioritized complex rule in $\mathcal{R}$ is empty. Hence, the runtime complexity of SSA is in $\mathcal{O}\left(|\mathcal{R}|^2\right)$.

For the example rule set $\mathcal{R}$ in Listing 1, the shunting vector $V_{\mathcal{R}_S}$ would be computed as follows:

$$V_{\mathcal{R}_S} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{matrix} \text{(for simple rule } R_1\text{)} \\ \text{(for simple rule } R_2\text{)} \\ \text{(for simple rule } R_4\text{)} \\ \text{(for simple rule } R_6\text{)} \end{matrix} \quad (2)$$

The rules R_1 and R_2, which are both simple rules, are the first and second rule in $\mathcal{R}$. Hence, they cannot conflict with any more highly prioritized complex rules, and therefore, their corresponding shunt bits are zero. Since rule R_4 does not conflict with the complex rule R_3 due to the different transport layer protocol check, R_4's shunt bit is also set to zero. Finally, R_6's shunt bit is set to one, because it conflicts with the more highly prioritized complex rule R_5. The resulting shunting vector `[0,0,0,1]` leads to fewer packet shunts than index-based shunting, since packets that first match rule R_4 can be processed entirely in hardware.

We now prove both the correctness and the *HSA-ideality* of shunting vectors computed by the SSA procedure.

Definition 1 (False Negative): A shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule R_i in $\mathcal{R}$ is false negative if $b = 0$ and if there exists a more highly prioritized complex rule R_j $(j < i)$ in $\mathcal{R}$ that conflicts with R.

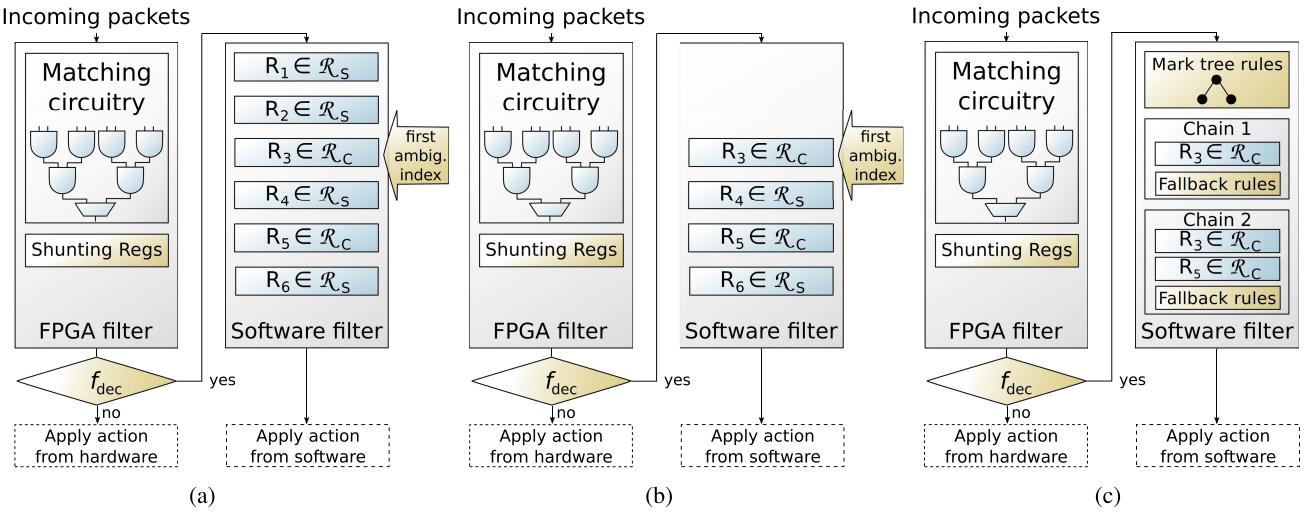

Fig. 4. Different strategies to implement the complex rule set $\mathcal{R}_A$ in the software filter. (a) Full set strategy. (b) Cut set strategy. (c) Interval strategy.

Definition 2 (Correctness): The shunting vector $V_{\mathcal{R}_S}$ for the rule set $\mathcal{R}$ is correct if it does not contain false negative shunt bits.

Theorem 1: The application of the SSA always results in correct shunting vectors.

Proof Sketch: Theorem 1 follows directly from Algorithm 1: for every bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule R_i, every more highly prioritized complex rule R_j $(j < i)$ in $\mathcal{R}$ is checked whether it conflicts with R_i. If such a complex rule exists, b is set to 1 and thus cannot be false negative.

Definition 3 (False Positive): A shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ that corresponds to the simple rule $R_i \in \mathcal{R}$ is false positive if $b = 1$ and if there does not exist a more highly prioritized complex rule R_j $(j < i)$ in $\mathcal{R}$ that conflicts with R_i.

Definition 4 (HSA-Ideal Shunting Vector): An HSA-ideal shunting vector $V_{\mathcal{R}_S}$ for the rule set $\mathcal{R}$ is correct and does not contain any false positive shunt bits.

Theorem 2: For any rule set $\mathcal{R}$, the application of SSA always computes an HSA-ideal shunting vector.

Proof Sketch: Let $V_{\mathcal{R}_S}$ be the SSA-computed shunting vector for a given rule set $\mathcal{R}$. The correctness of $V_{\mathcal{R}_S}$ follows from Theorem 1. Assume that $V_{\mathcal{R}_S}$ contains a false positive shunt bit $b = V_{\mathcal{R}_S}[\mathcal{R}_S(i)]$ for the simple rule R_i. Since b was set to 1 by the SSA procedure, there must exist an index j with $j < i$, such that the rule R_j is complex and conflicts with R_i. This contradicts the assumption that b is false positive.

We call a simple rule $R_i \in \mathcal{R}$ a *shunting rule* if the employed shunting technique (i.e., index-based or selective shunting) decides that a packet with *match_index* $= \mathcal{R}_S(i)$ must be shunted to the host for further processing. If this is not the case, we call R_i a *non-shunting rule*. Intuitively, it is desirable that many simple rules in $\mathcal{R}$ are non-shunting rules, as this will allow hardware-only processing if these rules are the highest prioritized matching ones.

In comparison to index-based shunting, selective shunting requires higher preprocessing times in case of a rule set update in order to compute the shunt vector. However, this effort is rewarded by much fewer packets having to be shunted to the host at runtime. This, in turn, leads to significantly higher packet processing rates, as we will show in our evaluation.

We point out that it is possible to even further reduce the number of shunted packets by installing the geometric reduction R_i^- of every complex rule R_i in the hardware matcher. In this case, the size of the hardware-computed result vector V_{res} increases from $|\mathcal{R}_S|$ to $|\mathcal{R}|$, as the hardware matcher also generates one result bit for the geometric reduction of every complex rule. Consequently, a packet P only has to be shunted if the most significant set bit $V_{\text{res}}[i^*]$ corresponds to a complex rule $R_{i^*} \in \mathcal{R}$. This approach requires modifications of our rule set-tailored matching circuitry in case of a rule set update, which is very time-consuming. In contrast, a software rule set update could be carried out in less than two seconds in most cases. Hence, we opted for the proposed shunting strategies.

VI. Software Filter

As described in Sections IV and V, the task of the software filter running on the host computer (`netfilter` in our example) is to classify every shunted packet which cannot be handled exclusively in hardware. However, simply installing only the complex rule set $\mathcal{R}_C$ in the software filter is not sufficient, since shunted packets P could still also match simple rules in $\mathcal{R}_S$. This is the case when P is not matched by any complex rule with a higher priority than the first matching simple rule. As a consequence, the software filter must be able to reproduce the hardware classification result iff the most highly prioritized matching rule is in $\mathcal{R}_S$ and not in $\mathcal{R}_C$. To cope with both shunting methods, we define the decision function f_{dec} to decide whether a packet needs to be shunted for this section. In the case of index-based shunting, this function is a comparison of *match_index* $\geq$ *shunt_index*. With selective shunting, it is instead checked whether $V_{\mathcal{R}_S}[\textit{match\_index}] = 1$. In this section, we present three different strategies how the rule set in the software filter can be organized to achieve this goal.

A. Full Set Strategy

The most straightforward way to setup the software filter, which we call the *full set strategy*, is to simply install the entire rule set $\mathcal{R}$. That way, forwarded packets will always traverse rules in the correct order until the first matching rule is found, as sketched in Figure 4a for the example rule set from Figure 1. This approach allows for quick rule updates, since only one rule in the rule set installed in the software filter has to be changed in addition to a possible update of the shunting policies on the FPGA. This strategy is simple, but has a major disadvantage: the software filter may process a large number of rules for every shunted packet, including simple rules. It thus repeats significant work already done in hardware. This can be particularly expensive as, in contrast with the full-parallel match in the hardware filter, the rules are commonly processed linearly in software packet filters.

B. Cut Set Strategy

The amount of redundant work that is done in software for shunted packets can be reduced with a slight modification. Let β be an index such that a shunted packet can never match a rule $R_i \in \mathcal{R}$ with $i < \beta$. In the case of index-based shunting, β is equal to *shunt_index*. For selective shunting, β is the index of the most highly prioritized simple rule $R_\beta \in \mathcal{R}$ with $V_{\mathcal{R}_S}[\mathcal{R}_S(\beta)] = 1$. We already know that no simple rule with an index less than β can match a packet that has been forwarded to the software filter—otherwise the packet would have been processed solely on the FPGA. For example, with index-based shunting, consider the rule set from Figure 1 and a packet P with *match_index* $= 3$. As *match_index* is equal to *shunt_index* (which is also 3 in the example), P will be forwarded to the software filter, which will superfluously once again test rules R_1 and R_2 against P. In order to avoid this potential extra work on the host system, the *cut set strategy* installs only those rules in $\mathcal{R}_C$ and in $\{R_j | R_j \in \mathcal{R}_S \wedge j \geq \beta\}$ in the software filter, as sketched in Figure 4b.

With the same HSA calculations that we used in selective shunting, this rule set can be further reduced, since the analysis exactly determines which rules of $\mathcal{R}_S$ the packet could match. Therefore, it is only necessary to install all rules in $\mathcal{R}_C$ and in $\{R_j | R_j \in \mathcal{R}_S \wedge V_{\mathcal{R}_S}[j] = 1\}$ in the software filter. We will call this improved variant *HSA cut set strategy*. This reduction produces correct results, regardless of whether index-based or selective shunting is used in the hardware matching unit.

In comparison to the full set strategy, both variants of the cut set strategy have higher rule update costs, as a potentially larger number of rules must be inserted or removed from the software filter in case of an update. Furthermore, the HSA cut set variant requires the computation of the shunting vector $V_{\mathcal{R}_S}$ in order to select those rules that must be installed in the software filter. However, our evaluation shows that the update effort clearly pays off in terms of classification performance, as fewer rules must be traversed by shunted packets.

C. Interval Strategy

The strategies described so far implement rule sets in the software filter that are agnostic to the partial classification

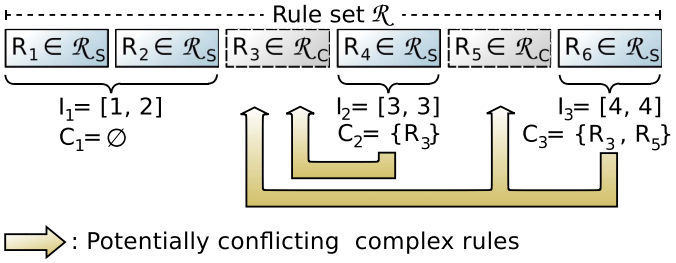

Fig. 5. Intervals in the rule set $\mathcal{R}$.

result tuple <*match_index*, $A_{\mathcal{R}_S,P}$> previously computed on the FPGA for every shunted packet P. This results in wasted effort on the software side and inflates the software-side rule set—also in case of the cut set strategy. To avoid the recomputation effort, the *interval strategy* relies on metadata handed over from the FPGA to the matching software when a packet is shunted, i.e., the match index and action tuple <*match_index*, $A_{\mathcal{R}_S,P}$>. Simply put, the goal of the interval strategy is that shunted packets should only be tested against a fraction of the complex rules $\mathcal{R}_C$ and none of the rules in $\mathcal{R}_S$ in software again.

The idea behind the interval strategy is that groups of consecutive simple rules $G_k = \{R_i, \ldots, R_{i+\alpha}\}$ in $\mathcal{R}$ can be mapped to intervals $I_k = [\mathcal{R}_S(i), \mathcal{R}_S(i+\alpha)]$. For instance, the simple rules from the example rule set in Figure 5 form three groups $G_1 = \{R_1, R_2\}, G_2 = \{R_4\}$, and $G_3 = \{R_6\}$, with the corresponding intervals $I_1 = [1, 2]$, $I_2 = [3, 3]$, and $I_3 = [4, 4]$. Each interval represents a range of match indices, which may be computed by the FPGA for an incoming packet P. If P is shunted to the host, then the *match_index* computed on the FPGA falls into exactly one of these intervals. The interval strategy exploits this fact by precomputing the chain of complex rules C_k for every interval I_k that could contain a more highly prioritized matching rule for a packet P whose hardware-computed *match_index* falls into interval I_k (i.e., P matches a simple rule in group G_k). In the example shown in Figure 5, C_1 is empty, since there are no complex rules in $\mathcal{R}$ that are more highly prioritized than the simple rules R_1 and R_2. In contrast, $C_2 = \{R_3\}$, as the complex rule R_3 is more highly prioritized than the simple rule R_4 and thus could match on packets that have been assigned to R_4 by the FPGA. Similarly, C_3 is set to $\{R_3, R_5\}$, as R_3 and R_5 are more highly prioritized than the simple rule R_6.

Now, whenever a packet P is shunted to the host, the FPGA driver fetches the <*match_index*, $A_{\mathcal{R}_S,P}$> tuple from the hardware. Then, the FPGA driver code on the host determines the index k of the interval I_k that contains the *match_index*. Before the actual `netfilter` packet classification starts, the index k, as well as the hardware action code $A_{\mathcal{R}_S,P}$, are written to the most significant 28 and least significant 4 bits of the `netfilter` `mark` field, which is a 32 bit metadata field attached to the packet P. With `netfilter` supporting tests on the `mark` field, we can use this information to achieve two goals: first, we want to limit the set of complex rules that must be tested in `netfilter` to only those that are more highly prioritized than the first matching simple rule.

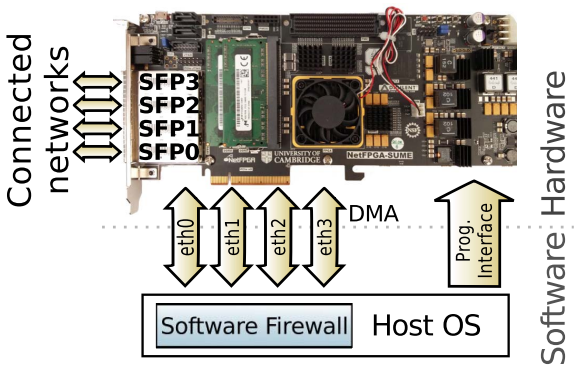

Fig. 6. Proposed structure of a HyPaFilter+ system. The host can be any COTS system capable of carrying the FPGA NIC.

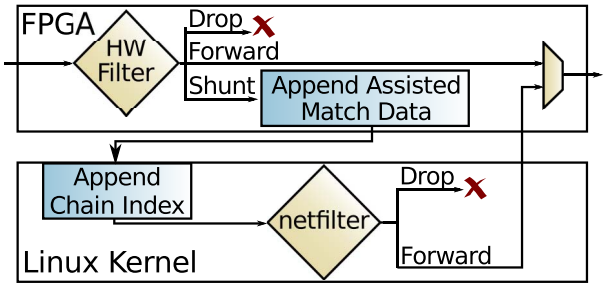

Fig. 7. Flow of packets through the HyPaFilter+ system.

Second, we want to apply the hardware-computed action $A_{\mathcal{R}_S,P}$ in `netfilter` *without* the need to re-traverse any simple rule in software if there is no match in $\mathcal{R}_C$.

To this end, the rules that are installed in `netfilter` for the interval strategy are generated as follows: the `netfilter` rule set starts with a sequence of rules which implement a binary search over the interval index k encoded in the most significant 28 bits of the `mark` field. This is done in order to quickly locate the chain of relevant complex rules C_k during the matching process, as sketched in Figure 4c. The generated rule set also contains each chain C_i as a linear list, which contains the complex rules that are mapped to interval I_i. Finally, the last rule in every chain C_i ends with a jump to a small set of fallback rules (one for each possible action), which use the least significant four bits of the `mark` field in order to apply the action $A_{\mathcal{R}_S,P}$ to the shunted packet P if no complex rule matches.

In comparison to the full set and cut set strategies, the interval strategy requires more complex preprocessing in case of a rule update, as the intervals for the complex rules have to be re-computed and communicated to the hardware driver. Furthermore, the `netfilter` binary search tree encoded in the filter rules must be re-generated. However, this strategy provides the best classification performance in software, as the number of traversed rules for each shunted packet P can be orders of magnitude smaller than in the full set and cut set strategies, as indicated by our evaluation. Furthermore, this approach does not require a change of the `netfilter` source code in order to use the hardware-computed matching information. Instead, we completely rely on existing `netfilter` match functionality to accelerate the software matching process.

VII. System Architecture

Our prototype system for the HyPaFilter+ system consists of two functional units. One part is a standard host system, used to run the software firewall and the toolchain for managing the system. This can even be an already existing firewall appliance which needs to be upgraded in terms of performance. This system is extended by the second part, a general purpose FPGA addon card, as shown in Figure 6. These units must provide a sufficient communication path for transferring data and settings between them. The utilized plug-in card is a suitable FPGA platform that can provide the required interfaces to both communicate with external Ethernet networks as well as acting as a regular network interface card in regard to the host system. It provides multiple network ports and can be plugged into a COTS system via PCI Express (PCIe). The card acts as the primary network interface connected to both the internal (e.g., LAN) and the external network (e.g., Internet). The hardware-based filtering is handled on the FPGA.

The host system carries the FPGA NIC and communicates with it via PCIe. The host runs the operating system where the back-end firewall `netfilter` with `iptables` is installed. It also supplies the tools to configure the FPGA, and provides a user interface for administrating HyPaFilter+.

The host and the FPGA card are connected through several communication channels. For quick and simple configuration settings, the host system is able to set and read predefined 32 bit registers on the FPGA via PCIe. Network traffic between FPGA and host is handled via *direct memory access (DMA)*. On the host side, a driver provides the functionality and interfaces so that the operating system can access the FPGA like a regular NIC. This is important since we do not want to rely on non-standard customizations to `netfilter` or other core components for HyPaFilter+ to work. By using a programming interface, the configuration of the FPGA can be updated. We used the Xilinx Vivado software toolchain to generate the FPGA configuration based on a given rule set.

A. System Operation

Incoming packets received from any connected network are first matched against the rules implemented on the FPGA. Based on the result and its validity with respect to the rule dependencies, packets are either dropped, forwarded directly (without interaction of the host system), or shunted to the host for further processing. Whenever a packet is shunted, the matching information—*match_index* and hardware action—is added to the packet in a driver-readable meta data field, which is needed for the interval strategy. For outgoing packets from the host system, the FPGA NIC acts like a standard NIC. Such packets, e.g., packets shunted and processed by the software firewall or packets generated by the host itself, are therefore sent out through the corresponding network interface without further analysis. The packet flow is sketched in Figure 7.

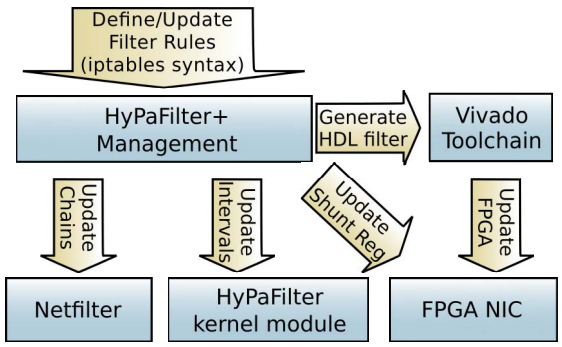

Fig. 8. HyPaFilter+ workflow with the management tool.

Note that in some rare cases, shunting can lead to packet re-ordering, if the rule set is configured to shunt only a part of the flow's packets, e.g., distinguished by additional header fields. Re-ordering is allowed for IP packets, but should be avoided due to, e.g., negative effects on the performance of TCP. However, rule constructions where this can occur are rather theoretical. Packets classified with rule sets that only test for fields examinable by the hardware matcher are either completely shunted or processed in hardware. In fact, in all real rule sets we analyzed during our evaluation, no packet re-ordering takes place.

The hybrid operation of HyPaFilter+ can bridge the delay that may occur when the hardware filter core needs to be updated. For this purpose, packets matching rules affected by the update are shunted until the change in the hardware becomes active. This allows to use hardware filters where updates are costly in terms of time.

The administrator needs to be able to manage the system without the need to understand the underlying complexity. In our implementation, we created a Python command line interface management tool. The general workflow for using HyPaFilter+ is shown in Figure 8.

B. Packet Data Path

The data flow through the FPGA can be shown in two layers. The underlying structure for general networking and communication tasks is based on the NetFPGA SUME pipeline [31]. The actual core which is responsible for filtering is embedded into this pipeline and connected via the AXI4 stream protocol. Internally, the HyPaFilter+ core uses a data bus width of 512 bits and runs at 180 MHz. Hence, the theoretically achievable throughput of 92.16 Gbit/s is enough to fully saturate all four 10 Gbit/s Ethernet ports. The NetFPGA SUME currently uses a bus width of 256 bits which is converted before and after the hardware core.

Packets coming into the hardware core are first distributed (cloned) into a classification path and a data path, with the latter being a simple FIFO queue of 64 kB. In the classification path, the *Header Parser* extracts relevant information from incoming packets. For a versatile operation, the header parser must take care of the data alignment due to VLAN tag-stacks or various variable-length headers. Therefore, it is implemented as a multi-stage non-blocking pipeline architecture. The preprocessed data is forwarded to the filtering module, which is generated by the management toolchain. After the classification, the decision is forwarded to the *Output Processing*, where the determined action is executed: DROP (read from FIFO and discard), FORWARD, or SHUNT by adapting the output port field in the packet's metadata.

As described in Section IV, the matching logic is able to classify packets in constant time. Since the hardware filtering logic contains no components that could cause a data-pipeline stall, it is clear that the HyPaFilter+ hardware core is never the limiting factor for raw data throughput in this setup. The hardware filter core is able to extract and classify incoming packets against a variety of parameters like IP addresses, protocol fields, MAC addresses, and port fields.

Previous work has shown that the resource utilization of typical rule sets on FPGAs can be significantly reduced by including the actual rule set in the logic optimization process, rather than using a generalized filtering logic [10], [30]. As firewall rule sets in general are not static, an FPGA's reconfigurability allows us to exploit this potential in practical applications. The HyPaFilter+ prototype combines such a rule set tailored hardware filter with a generic software filter residing on the host. However, HyPaFilter+ does not strictly rely on this type of specialized hardware filter. It could also utilize other hardware matchers (e.g., a TCAM), as long as the the required matching information can be extracted from the result and used for selective shunting of packets.

VIII. Evaluation

The main goal of the HyPaFilter+ architecture is to increase the achievable throughput of software packet classification systems that support complex rules. Therefore, we investigate in our evaluation the extent to which the classification performance of a representative software filter can be improved when used with HyPaFilter+, and how the performance varies with rule set size and structure. Furthermore, we evaluate in detail the impact of the two presented shunting techniques on the number of packets that must be processed on the host system. Also, we analyze the performance of the different strategies to organize the rules in the software filter in terms of number of traversed rules and rule set update time. Finally, we measure the additional packet processing latency induced by the FPGA.

A. Measurement Setup

Our measurement setup consists of two dedicated machines, each of which contains an Intel E3-1270 CPU, 16 GB RAM, a dual-port 10 Gbit/s NIC and runs CentOS 6.6. One machine was configured to send small packets as fast as possible (the sender), while the other machine was used as a packet sink (the receiver). Here, all sent packets carry five arbitrarily chosen bytes as payload. Using small packets causes a much higher workload on the classification engine, as more packet headers need to be analyzed in the same time frame compared to using large packets at the same data rate. For our evaluation, we set up a typical bridging firewall scenario as shown in Figure 9. The sender and receiver hosts are connected to the HyPaFilter+ system via optical fibre. We counted the number of packets received by the MAC-Core `MAC0` on the NetFPGA and those arriving on the network interface of the receiver.

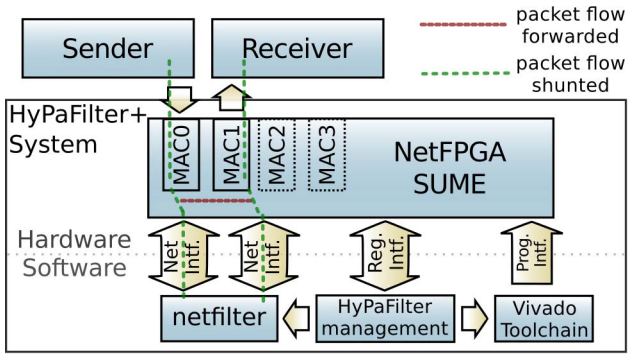

Fig. 9. Evaluation setup showing the relevant components. Traffic is generated on the sender and directed through the bridging HyPaFilter+ firewall.

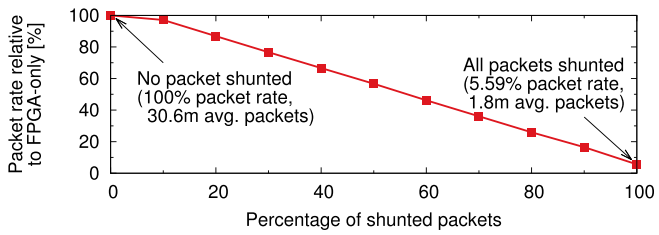

Fig. 10. Impact of shunting packets without firewalling.

The HyPaFilter+ system consists of the following relevant components: Intel Xeon E3-1230 based host, 16 GB RAM, NetFPGA SUME PCIe card, Debian 8.3, `netfilter` framework and `iptables` v1.4.21, Xilinx Vivado 2014.4, and the HyPaFilter+ management tools. The hardware filter core is integrated into a data pipeline based on the reference NIC project of the NetFPGA SUME release 1.0.0.

B. Impact of Packet Shunting

In our first experiment, we measured the impact of shunting packets to the software. As processing packets directly in hardware without any software interaction provides the lowest latency and highest packet rate, we expect to achieve the best packet rate if no packets are shunted at all. If the fraction of packets that are shunted to the host increases, a drop in the obtained packet rate is to be expected. For this test, the FPGA NIC was configured to forward a certain number of the packets directly to the target, while the other packets were shunted to the host. To exclude additional processing overhead on the host system, all shunted packets were directly software-bridged to the outgoing interface without software firewall interaction.

We compared the number of ingress packets on the FPGA to the number of packets received at the receiver. Each data point shows the average packet rate of ten test runs, each lasting for 20 seconds. The standard deviation is too small to be visible. The average number of ingress packets arriving at the HyPaFilter+ network interface in each test run before any classification is 30 million packets in 20 s. Figure 10 shows the results of this experiment. When all packets are forwarded by the FPGA, the test setup is capable of operating at line speed. With more packets being shunted to the software, the performance drops continuously. When all packets are shunted, the NetFPGA SUME basically acts as a simple NIC. In this case, only 5.59% of the packets can be handled by the evaluated system. This demonstrates that in a hybrid system like the proposed one, FPGA-to-host communication is expensive and should be avoided in order to reach line speed performance. In comparison, the detection and output queue assignment of a shunted packet extends the processing pipeline by just one clock cycle, which has no measurable effect on the throughput or packet rate.

C. Test Rule Sets

To evaluate the classification performance of the HyPaFilter+ system under repeatable conditions, we used several different rule sets, synthetic as well as real-world ones. All rules had been given the action ACCEPT, so the number of dropped packets can be regarded as the packet loss solely due to the architecture. As we only evaluate stateless classification performance, no state handling is performed and the network protocol of a packet has no influence on the processing speed. For reasons of simplicity, all rules were considered as UDP rules. The initial case implements all rules as simple rules on the hardware. The number of rules that could be matched within a single clock cycle on the FPGA was 1 100.

We generated 10 different synthetic UDP rule sets of 1 100 rules each with ClassBench [32], which are publicly available at [33]. Also, we were granted access to three confidential real-world rule sets by one customer of the genua company. These rule sets were transformed to be used in our setup, which means that we only used the parameters important for measuring the classification performance. Due to the size limitation, only the first 1 100 rules of each set were used in our first experiments. In Section VIII-I, we add more pipeline stages and use approximately the original rule set size.

For each test, the rule set was translated by the HyPaFilter+ management tool, integrated in our hardware firewalling module, embedded in the NetFPGA SUME pipeline and afterwards synthesized and implemented into an FPGA configuration bitfile. The resource utilization of the Virtex 7 690T FPGA is similar for all test rule sets at about 9% of the available flip-flops (FF) and 15% of the look-up tables (LUT). The HyPaFilter+ core uses less than 1% FFs and about 2% LUTs.

To measure the impact of changes or occurrences of complex rules to the rule set, we modified the rule set during the test as follows: at k positions ($k \in \{1, 5, 10, 15, \ldots, 50\}$) equally distributed over the rule set, the simple rules at these positions were augmented with a string matching and probabilistic matching part to enforce shunting:

```
-m string --algo bm ! --string BAD -m \
statistic --mode random --probability 0.99
```

This is intended to show the possibility for demanding worst-case complex rules. To evaluate the effect of the properties of these complex rules, we repeated the performance tests with “best-case complex rules” that had no complex operation (i.e., they artificially enforced shunting without additional

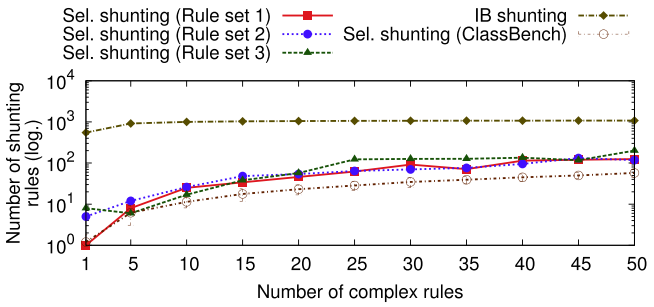

Fig. 11. Number of shunting rules, index-based (IB) vs. selective (sel.) shunting.

processing steps in software). We found an negligible average performance increase of 0.036 pp ($\sigma = 0.15$). Therefore, the nature of the complex rules is not important for our evaluation.

For all rule sets, we used ClassBench's *trace_generator* to generate a trace of 100 000 packet headers that were uniformly distributed over the corresponding rule set. These rule sets and corresponding traces were used in the following experiments.

D. Shunting Technique Effectiveness

In this section, we evaluate the effectiveness of the two shunting techniques introduced in Section V in terms of the number of shunting rules and the number of actually shunted packets. Recall that a packet P is shunted to the software filter if the most highly prioritized matching simple rule is a shunting rule. Hence, if all simple rules in $\mathcal{R}$ are shunting rules, every incoming packet will be processed in software. Likewise, if there are no shunting rules in $\mathcal{R}$, the entire traffic can be processed solely in hardware at line speed. We measured these two quantities by performing the following experiment for every synthetic and real rule set:

1) Load the initial simple rule set.
2) Add complex part to k ($k \in \{1, 5, 10, \ldots, 50\}$) rules at equally-spaced positions from the initial rule set.
3) Match the corresponding trace file using index-based and selective shunting against the installed rule set.
4) Count the numbers of shunting rules and shunted packets.

The numbers of shunting rules and shunted packets are shown in Figures 11 and 12, respectively. It can be seen that the index-based shunting technique results in shunting of a relatively large number of packets—all packets with *match_index* greater than or equal to the lowest modified rule index. Selective shunting reduces the number of shunting rules significantly. Of course, the actual number of selectively shunted packets depends on the rule set characteristics, i.e., the number of simple rules that conflict with more highly prioritized complex rules. Figure 11 reveals that ClassBench-generated rule sets are nearly independent regarding header space, leading to an almost equal number of shunting rules compared to the number of complex rules. Hence, selective shunting performs particularly well.

However, the real rule sets are more diverse with regard to their intention and have more header space variation.

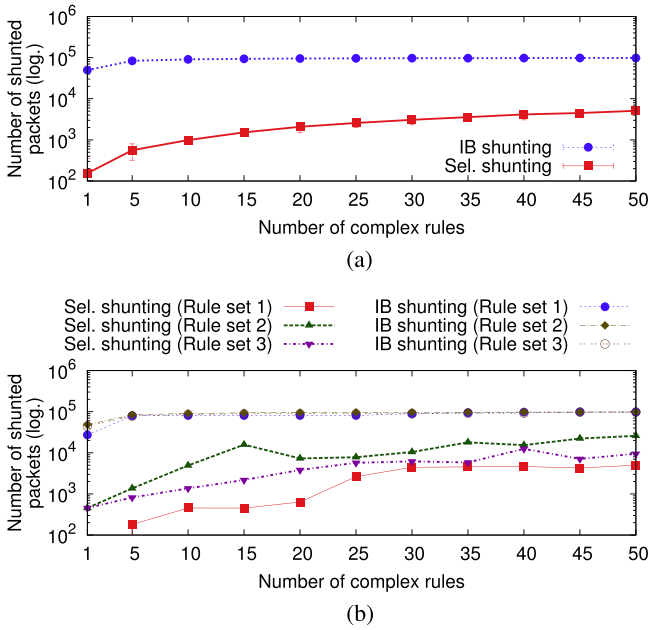

Fig. 12. Number of shunted packets, index-based (IB) vs. selective (sel.) shunting. (a) Synthetic ClassBench rule sets (averaged, with std. dev.). (b) Real rule sets.

Depending on which rules are altered and extended by a complex part in the test, this can result in fewer shunting rules, even when more complex rules are used. Nevertheless, also in the case of real-world rule sets, the number of non-shunting rules in case of selective shunting is still at least one order of magnitude greater than in the case of index-based shunting, as shown in Figure 11. With index-based shunting, the number of shunting rules is identical for all rule sets, which is why only one line is shown. Since the packets generated by ClassBench's trace generator activate the rules with an approximately uniform distribution, the number of shunted packets shows a distribution that is similar to the number of shunting rules, as confirmed by Figure 12. The figure indicates that selective shunting significantly reduces the number of software-processed packets and is therefore superior to index-based with regard to the overall system performance.

E. Software Strategy Comparison

Although the use of shunting strategies, especially of selective shunting, can significantly reduce the workload on the host system, every shunted packet must still be processed in software. As software classification is expensive in comparison to hardware-only processing, we presented different rule set organization strategies in Section VI that accelerate the rule set traversal in software. In this section, we evaluate these strategies in terms of the number of rules that must be traversed to classify all shunted packets. Since the rules are mostly traversed linearly in software, this provides a direct indication of the amount of additional work performed on the host system. Here, the examined packets are identical to the shunted packets in Section VIII-D using index-based shunting, so the software workload is identical in all test runs. For every shunted packet (for a specific number of complex rules and the

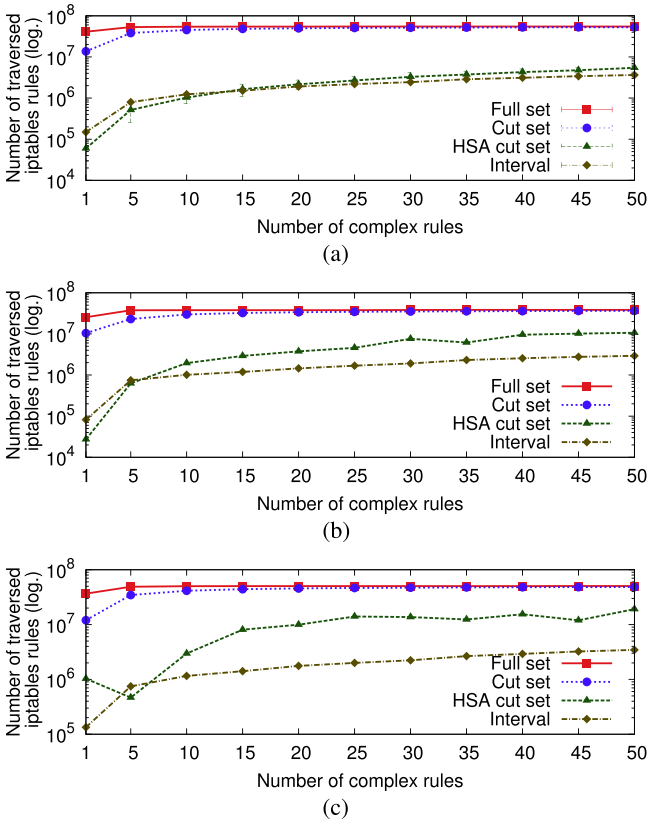

Fig. 13. Number of traversed software rules for shunted packets using different strategies. Real rule set 2 omitted as results are similar to rule set 1. (a) Synthetic ClassBench rule sets (averaged, with std. dev.). (b) Real rule set 1. (c) Real rule set 3.

selected strategy), we determined the number of rules (the *rule path length*) traverses in software until it was fully classified. The sums of these path lengths are shown in Figure 13.

The average results for the ten synthetic ClassBench rule sets are shown in Figure 13a, including their respective standard deviations. As mentioned in Section VI, maintaining the complete rule set in the software filter—as it is the case for the full set strategy—leads to a high redundancy. This is reflected in the figure, as the full set strategy leads to the highest number of traversed rules. The cut set strategy reduces this redundancy, but its benefits are limited to cases where complex rules do not appear with a high priority—otherwise, it performs close to the full set strategy. Further reducing the cut set with HSA, as explained in Section VI, can significantly reduce the number of traversed rules, especially when testing with mainly independent rules, as it is the case for synthetic ClassBench rule sets. Here, this strategy can even perform better than the interval strategy for up to five complex rules.

Still, of all strategies, the interval strategy generally provides the fastest software matching by using the hardware matching information to traverse a binary search tree. With real rule sets and more overlap in the header space of the rules, the differences become clearer as it can be seen in Figures 13b and 13c. The results clearly demonstrate that the additional processing efforts for the interval or HSA cut set strategies are worthwhile by resulting in significantly lower workload in the software classification engine.

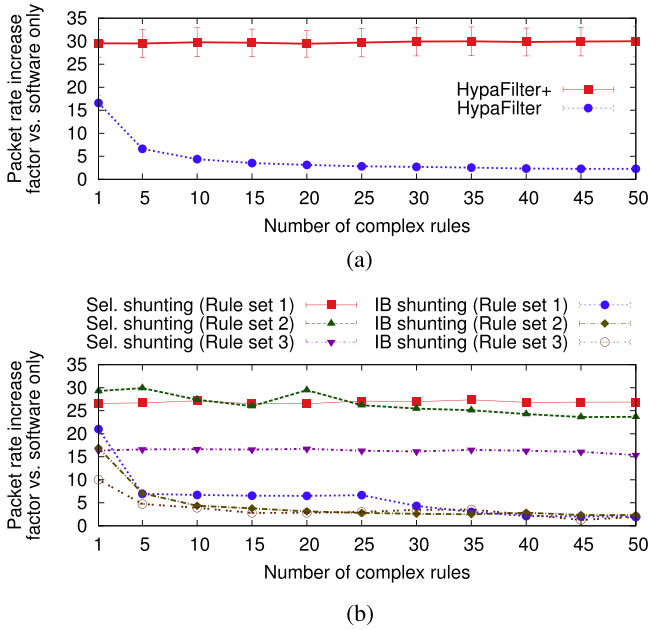

Fig. 14. Speedup of HyPaFilter+ compared to software-only with index-based (IB) and selective (sel.) shunting. (a) Synthetic ClassBench rule sets (averaged, with std. dev.). (b) Real rule sets (averaged over 10 runs, with std. dev.).

F. Throughput Measurements

Up to this point, we evaluated the different shunting techniques and approaches to software rule set organization in isolation. In this experiment, we put the pieces together and performed throughput measurements using both index-based shunting and selective shunting on our hardware platform. We used the interval strategy to organize the rules in the software filter, as this strategy proved to be the most efficient one with respect to classification performance. The measurement method is identical to the packet loss test in Section VIII-B. We compared the packet throughput rate of HyPaFilter and HyPaFilter+ against a reference measurement of the equivalent software firewalling setup, using `netfilter` and the NetFPGA running as a simple NIC. The procedure for ($k \in \{1, 5, 10, 15, \ldots, 50\}$) is described in the following and was repeated for all test rule sets.

1) Implement the initial test rule set onto the FPGA and set shunting vector to match everything in hardware.
2) Execute test run $k = 0$ without any update.
3) Modify k rules at equally-spaced positions from the initial rule set and add the complex part.
4) Update the software filter using the interval strategy.
5) Calculate and set shunting registers on the FPGA.
6) Execute the test run.
7) Re-initialize and repeat from step 3.

Figure 14a shows that the index-based shunting of HyPaFilter—while still faster than software-only—achieves its best results as expected when only few complex rules are used. This is mainly due to the fact that with equally-spaced complex rules, an increased number of complex rules results in such rules appearing at higher priorities in the rule set. Therefore, a large amount of traffic is shunted. On the contrary, since

only few packets need to be shunted, the selective shunting of HyPaFilter+ is able to maintain a constant 30-fold packet-rate increase compared to software-only processing.

The results in Figure 14b confirm that these numbers are still valid for real rule sets. As explained in Section VIII-D, in comparison to synthetic rule sets, the real rules have more diverse characteristics. This causes the greater variation and non-monotonic behaviour during this test.

G. Network Latency

While the packet classification rate is the most interesting parameter to measure for evaluation, the additional latency that is added by security appliances can be a major issue for certain applications, such as data centers. Our network latency measurement splits into two parts: the additional delay of the HyPaFilter+ hardware core in the NetFPGA SUME pipeline and the actual delay that can be seen on network packets. The internal additional delay in the FPGA is 24 clock cycles. With a clock rate of 180 MHz, the core therefore adds an additional delay of 133 ns compared to the NetFPGA SUME in NIC operation. In order to check for the overall network latency imposed by the HyPaFilter+ system, the round-trip time (RTT) was measured with `ping`, sending 50 packets per test. While a direct connection between sender and receiver (without the NetFPGA SUME) shows a one-way latency of 51 μs ($\sigma = 3.2$ μs), with the HyPaFilter+ system present and forwarded packets only we saw a tolerable increase to 52 μs ($\sigma = 5.4$ μs). For packets shunted through software without any firewall interaction it further increased to 73 μs ($\sigma = 3.5$ μs). The highest average delay of 96 μs ($\sigma = 7$ μs) occurred with shunted packets and an active synthetic software rule set of 1 100 rules loaded into `netfilter`.

H. Rule Set Update Delay

Another interesting parameter is the time required to update the rule set using the different strategies. We therefore measured the time for modifying rules and updating the *shunt_index* or shunting vector registers in the FPGA. Taking into account the increasing calculation effort with an increasing number of inserted rules, the delays for the insertion were determined for consecutive rule insertions. The software strategy updates were first tested with the inexpensive index-based shunting method in order to better distinguish between the delay for the strategies and the calculation of the shunting vector for selective shunting. The following tests have been conducted, taking the time for all steps:

a) for the full set strategy: update a single rule with `iptables` and set *shunt_index*,
b) for the cut set strategy: truncate the rule set, insert and load this set with `iptables-restore`, set *shunt_index*,
c) for the HSA cut set strategy: calculate the shunting vector and create the corresponding rule set, insert and load this set with `iptables-restore`, set the shunting vector registers on the FPGA,
d) for the interval strategy with index-based shunting: calculate intervals, insert the chained rule set with `iptables-restore`, update the driver and set *shunt_index*,
e) for the interval strategy with selective shunting analysis: calculate intervals, insert the chained rule set with `iptables-restore`, update the driver, calculate the shunting vector and set the shunting vector registers.

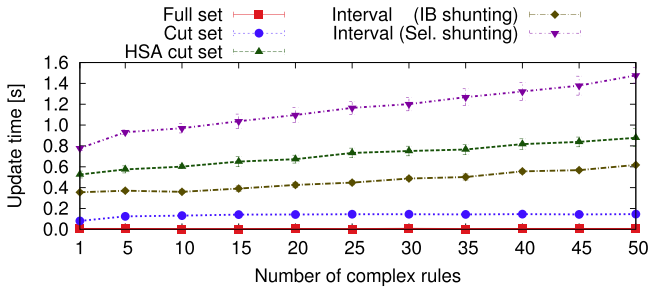

Fig. 15. Rule set update latency for consecutive insertion using different software strategies and index-based (IB) or selective (Sel.) shunting (averaged, with std. dev.).

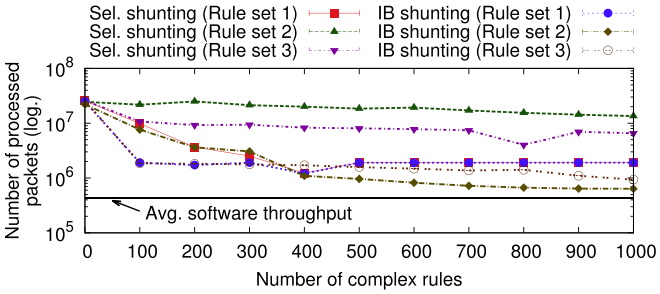

Fig. 16. Worst-case scenario speedup using index-based (IB) and selective (sel.) shunting.

Figure 15 shows the result of this test, as an average of all 13 test rule sets used in the evaluation. Setting one register on the FPGA from the host alone takes 1 μs. Figure 15 confirms that even the demanding updates of the interval strategy with selective shunting could be carried out with a tolerable delay.

Updating the logic optimized rule set on the FPGA was only necessary once for every test rule set. The bitfile generation took about 45 minutes on the HyPaFilter+ evaluation host. The FPGA configuration with the Xilinx configuration tool `xmd` (via USB-based JTAG) finished in 17.38 s.

I. Scalability

The HyPaFilter+ approach can also be used for larger rule sets than shown so far. The (interchangeable) hardware matching unit can be configured in a pipelined layout which we successfully tested with five stages of 1000 rules each. This increases the latency by 13 clock cycles (72.2 ns at a clock rate of 180 MHz) per stage, without affecting the throughput.

To evaluate the behaviour of the system if a greater number of rules is placed into the FPGA and a greater share of the hardware rules is adapted to complex processing, we repeated our former tests with 5000 rules and up to 1000 complex rules. In this case, the number of shunting rules is, on average, 68.6% greater than the number of complex rules. As with this

setup a larger fraction of the rules are changed to complex rules, the system's performance will approach the software-only firewall level, as shown in Figure 16. At 1000 complex out of a total of 5000 rules, the speedup vs. software still reaches 19.7x on average.

Regarding update latency, the critical step is the calculation of the HSA vector, which has a complexity of $\mathcal{O}\left(|\mathcal{R}|^2\right)$. In the worst case, this operation took up to 18.3 s in this evaluation. All other update operations (i.e., registers) scale with $\mathcal{O}(|\mathcal{R}|)$.

IX. Conclusion

We introduced HyPaFilter+, a hybrid packet classification approach, which combines the parallel matching capabilities of specialized hardware with the extensive matching semantics of software packet filters. HyPaFilter+ accomplishes this task by partitioning the implemented packet processing policy into a simple and a complex part, where the simple part can be handled directly in hardware and the complex part is installed in the software filter. Incoming network packets are first processed in hardware and are shunted to the software filter only in cases where complex processing is required. We present a novel strategy how the software-implemented part of the rule set can be organized in order to reuse matching information from the hardware. This strategy can be used on top of `netfilter` and does not require changes of the `netfilter` source code. The actual hardware filter is not limited to our evaluation example. It can be any suitable architecture that provides its matching result to the host system, such as a P4 [34] based hardware switch. We further showed how to leverage geometric rule representations to significantly reduce the need for shunting packets to the software. With the addition of selective shunting, our evaluation demonstrates that also with real-world rule sets and a large number of complex rules, software processing can be avoided for major parts of the traffic. As a result, our HyPaFilter+ prototype based on a combination of a NetFPGA SUME FPGA and a Linux host system demonstrates up to 30-fold increases in the achievable throughput over a software-only approach.

References

[1] *The Netfilter.Org Project*. Accessed: Jun. 10, 2017. [Online]. Available: https://www.netfilter.org

[2] *OpenBSD Packet Filter*. Accessed: Jun. 10, 2017. [Online]. Available: http://www.openbsd.org/faq/pf/

[3] *IPFW Firewall*. Accessed: Jun. 10, 2017. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?ipfw

[4] *Genugate Firewall*. Accessed: Jun. 10, 2017. [Online]. Available: https://www.genua.de/en/solutions/high-resistance-firewall-genugate.html

[5] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. FPGA*, Feb. 2009, pp. 219–228.

[6] D. Liu, B. Hua, X. Hu, and X. Tang, "High-performance packet classification algorithm for many-core and multithreaded network processor," in *Proc. CASES*, Oct. 2006, pp. 334–344.

[7] Y. Qi *et al.*, "Towards high-performance flow-level packet processing on multi-core network processors," in *Proc. ANCS*, Dec. 2007, pp. 17–26.

[8] W. Jiang and V. K. Prasanna, "A FPGA-based parallel architecture for scalable high-speed packet classification," in *Proc. ASAP*, Jul. 2009, pp. 24–31.

[9] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *Proc. HOTI*, Aug. 2012, pp. 1–8.

[10] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt, "MPFC: Massively parallel firewall circuits," in *Proc. LCN*, Sep. 2014, pp. 305–313.

[11] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multi-layer packet classification with graphics processing units," in *Proc. CoNEXT*, Dec. 2014, pp. 1–12.

[12] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. SIGCOMM*, Aug. 2013, pp. 99–110.

[13] K. Accardi, T. Bock, F. Hady, and J. Krueger, "Network processor acceleration for a linux* netfilter firewall," in *Proc. ANCS*, Oct. 2005, pp. 115–123.

[14] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[15] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore, "HyPaFilter—A versatile hybrid FPGA packet filter," in *Proc. ANCS*, Mar. 2016, pp. 25–36.

[16] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, Apr. 2012, pp. 113–126.

[17] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[18] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.

[19] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. HOTI*, Aug. 1999, pp. 1–9.

[20] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. SIGCOMM*, Aug. 2003, pp. 213–224.

[21] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. SIGCOMM*, Aug. 1998, pp. 203–214.

[22] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. SIGCOMM*, Aug. 2001, pp. 199–210.

[23] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. SIGCOMM*, Aug. 1999, pp. 135–146.

[24] S. Hager, S. Selent, and B. Scheuermann, "Trees in the list: Accelerating list-based packet classification through controlled rule set expansion," in *Proc. CoNEXT*, Dec. 2014, pp. 101–108.

[25] D. Qunfeng, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," in *Proc. SIGMETRICS*, Jun. 2007, pp. 253–264.

[26] A. X. Liu, E. Torng, and C. R. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *Proc. INFOCOM*, Apr. 2008, pp. 176–180.

[27] M. Chen *et al.*, "Using NetFPGA to offload linux netfilter firewall," in *Proc. 2nd North Amer. NetFPGA Develop. Workshop*, 2010, pp. 1–7.

[28] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400G+ packet classification," in *Proc. HPSR*, Jun. 2012, pp. 1–6.

[29] A. Fiessler, S. Hager, and B. Scheuermann, "Flexible line speed network packet classification using hybrid on-chip matching circuits," in *Proc. HPSR*, Jun. 2017, pp. 1–8.

[30] S. Hager, D. Bendyk, and B. Scheuermann, "Partial reconfiguration and specialized circuitry for flexible FPGA-based packet processing," in *Proc. ReConFig*, Dec. 2015, pp. 1–6.

[31] J. Lockwood *et al.*, "NetFPGA—An open platform for gigabit-rate network switching and routing," in *Proc. MSWiM*, Jun. 2007, pp. 160–161.

[32] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

[33] S. Hager. *Hypafilter Rule Sets*. Accessed: Sep. 4, 2017. [Online]. Available: http://hardfire.de/rule-sets

[34] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

Andreas Fiessler received the master's degree in electrical engineering and information technology, Humboldt University of Berlin, Germany, where he is currently pursuing the Ph.D. degree. He is also a Security Researcher with genua GmbH, Kirchheim, Germany. He is involved in the research project HARDFIRE, which aims at building an FPGA-based firewall. His research topics focus on firewalling, hardware security, and reconfigurable logic.

Claas Lorenz received the master's degree in computer science from the University of Potsdam, where he is currently pursuing the Ph.D. degree. He is also a Security Researcher with genua GmbH, Kirchheim, Germany. Since 2015, he has been a German Network Security Specialist with genua GmbH in the SARDINE Project, which is funded by the German Ministry of Education and Research. His research interests include firewalling, SDN/NFV, and formal security verification.

Sven Hager received the M.S. degree in computer science from Heinrich Heine University Düsseldorf, Germany, in 2013. He is currently pursuing the Ph.D. degree with the Computer Engineering Group, Humboldt University of Berlin, Germany. He is active in the field of network packet classification. His research interests include classification algorithms, hardware architectures for packet processing, and rule set transformations.

Björn Scheuermann He received the degree in mathematics and computer science and the Ph.D. degree in 2007. After holding professorships in Düsseldorf, Würzburg, and Bonn, he joined Humboldt University in 2012. He is currently a Full Professor and the Chair of computer engineering with the Humboldt University of Berlin, Germany.The focus of his scientific work is on performance, design, and security aspects of computer networks, where he is involved in, for instance, wireless communications, privacy and anonymity, and hardware design.

Andrew W. Moore (M'02) is currently a Reader in systems with the Computer Laboratory, University of Cambridge, U.K., where he is also part of the Systems Research Group working on issues of network and computer architecture. His research interests focus upon enabling open-network research and education using the NetFPGA platform. Other research pursuits include low power, energy-aware, networking, and novel network and systems datacenter architectures.