Dynamic Packet-filtering in High-speed Networks Using NetFPGAs

Felix Engelmann, Thomas Lukaseder, Benjamin Erb, Rens van der Heijden, Frank Kargl

Institute of Distributed Systems
University of Ulm
Albert-Einstein-Allee 11
89081 Ulm
felix.engelmann@sfz-bw.de, thomas.lukaseder@uni-ulm.de, benjamin.erb@uni-ulm.de,
rens.vanderheijden@uni-ulm.de, frank.kargl@uni-ulm.de

***Abstract*—Computational power for content filtering in high-speed networks reaches a limit, but many applications as intrusion detection systems rely on such processes. Especially signature based methods need extraction of header fields. Hence we created an parallel protocol-stack parser module on the NetFPGA 10G architecture with a framework for simple adaption to custom protocols. Our measurements prove that the appliance operates at 9.5 Gb/s with a delay in order of any active hop. The work provides the foundation to use for application specific projects in the NetFPGA context.**

I. Introduction

Modern network applications such as video streaming or cloud-based backup services constantly require more traffic, leading to increasing bandwidth for links. To accommodate this, the Ethernet specifications were raised up to 100 Gb/s [1] which outperforms most busses on motherboards for I/O connectivity.

For network debugging or intrusion detection it is useful to peek into the flow of a link and capture some packets for further processing or inspection. Other use cases of a packet filter include the collection of meta-data and statistical evaluation. When filtering a few packets, all the packets passing through the link have to be inspected for the filter to function correctly. A common architecture consists of a network interface controller (NIC) attached to a general purpose CPU. In this setup, the double link throughput has to be handled by the motherboard (to and from the NIC). This is possible for gigabit throughput, but can not yet be achieved at 10 G speeds with commodity hardware.

To enable extensive filter capabilities for high-speed links as well, we developed a packet filter that directly processes data in hardware. Using the field-programmable gate array (FPGA) of a so-called NetFPGA platform, a throughput of 40 Gb/s is possible. The filter card can then be attached to a link with no influence on the transmission. It facilitates parsing arbitrary byte-oriented protocols and detects packets by specified header fields matching one of several rules on multiple layers.

By default, filtered packets are forwarded and additionally duplicated to send them to a secondary diagnostic interface. This interface captures and analyses the packets using lightweight hardware. For firewall applications, this behaviour can easily be changed to drop packets or forward them to the capture interface.

We evaluate our implementation in order to verify line-speed capability, which includes none or minimal obligatory influence on the packet flow. This is split into latency and throughput.

II. Related Work

Many authors have worked on implementing packet analysis in hardware in the past [2], [3], [4], [5], [6], [7], often with applications in intrusion detection. In intrusion detection, a main challenge is not just the analysis of network headers, but also content inspection. To efficiently implement content inspection, different metrics can be used, including the added latency by the system, the throughput of the network and the throughput of the hardware. Previous work has mainly focused on the implementation of so-called signature-based network intrusion detection systems, which aim to detect known attacks using known communication patterns, typically expressed by regular expressions on the content.

Hutchings et al. [2] describe a hardware implementation of a signature-based network intrusion detection system using FPGAs. Their focus is on string matching using regular expressions and they note a staggering improvement by a factor of up to 600 for large patterns. The authors implement their matching using Non-deterministic Finite Automatas (NFAs) to model the syntactic elements of their expressions to automatically convert arbitrary regular expressions to their hardware implementations. They compare their implementation against an open-source NIDS, implemented in software. The throughput reached by their system is fairly constant in the case of a hardware implementation, providing around 300-400 kB/s for a 1MB file in chunks of 1kB, and up to 870 for a 10MB file in chunks of 16kB, adding a latency of up to1.2ms and 7.38ms, respectively. Compared to their work, our focus lies on much higher throughput (up to 1GB/s), where directly using regular expressions is not feasible. However, we note that our filters can be used to identify individual network

flows, which can then be analyzed by their regular expressions module.

Aldawairi et al. [3] also discussed the topic of regular expression matching, and showed an accelerator that uses finite state machines stored in RAM, attached to the FPGA, in order to improve the speed at which packets can be matched against regular expressions. However, they have not analysed their system in a real-world setting using network traffic. Instead, they focussed exclusively on the potential throughput in terms of computational overhead. In addition, we note their idea introduces a new bottleneck: the bandwidth of the bus to the RAM. They also discuss previous work, in particular the approach by Sourdis & Pnevmatikos[4], which parallelizes different matching procedures to perform fast regular expression matching. They also discuss the work by Gokhale et al. [5], which presents Grandidt, which focusses on reconfigurability without reconfiguration of hardware. Compared to all these approaches, our focus is on the second type of matching rules described by these works – not string matching, but protocol layer analysis.

Unlike the previous approaches, our goals are aimed at a packet filter that provides maximal network throughput and minimal latency. This filter will dynamically analyse network headers and can forward the filtered packets to an advanced filter, which can then perform arbitrary intrusion detection procedures or other processing. The advantage of our approach over the previous work is twofold: first, filtering packets this way is much more efficient in terms of network throughput, network latency and FPGA size; second, our approach does not force the implementation of all intrusion detection rules into regular expressions.

Kobiersky et al. [7] developed an FPGA based header analyser which is realised by a finite-state machine. In their workflow they describe the protocols of the stack in XML syntax and translated these to HDL. This process is done by first creating an FSM which digests the input byte wise. Because this needs too many cycles, they parallelise it into a multi-char FSM by finding common transitions in the original automatons. The additional conversion process and the complexity with wider busses could not fit to our use case. Also their architecture lacks the ability to change filter values at runtime.

We base our approach on the work by SCOTT [6], who developed a module for the 1G NetFPGA data flow which is capable of matching the first six 64 bit words to a static filter. In [6], the author introduces a content of the filter that is assembled from a human readable rule specification on the host of the card. It can then be transferred to the NetFPGA at runtime and takes effect immediately. Because the bit string filter is created in software, new protocols can easily be implemented without synthesising the FPGA firmware. However, the fixed offsets of the filter limit its capabilities to a tree shaped protocol stack, of which only one path can be filtered, because all previous header lengths are necessary to determine the offset. In an environment where a header can be embedded into several different protocols, only one combination per port can be filtered. While this might be sufficient for debugging purposes, it can not be used in an IDS setup with multiple filter rules to match.

III. IMPLEMENTATION

Our approach extends the capabilities of the filter developed by SCOTT by abstracting from any static offsets of header fields. Hence, our approach transparently handles dynamic offsets of known protocols. This is achieved by enhancing the hardware and adding the semantics of protocols. So the headers can be parsed on the FPGA and the offsets to the fields of interest are computed dynamically.

The basic concept of our filter module for the 10G NetFPGA platform is based on the work of SCOTT. It consists of two parts. One is used for buffering and forwarding packets to the right output ports and another more involved part, which is the actual parser and filter. The path through the entire FPGA, including our filter module, is depicted in Figure 1.

A. Environment

For the buffering and forwarding framework, it is crucial to perform with minimal impact on the data flow. The buffer is a standard fall-through FIFO buffer with the width of one word and a total size of 2048 bytes. It is directly attached to the input interface of the module. To avoid overflows, it can stall the input. This FIFO buffer stores as many words as needed to determine the output of the packet. As soon as the destination is made available in a second small FIFO buffer, data can be forwarded.

The output handler monitors the destination buffer. As long as there is an entry in it, the next packet from the data buffer is forwarded to the indicated port. This process allows a pipelined operation, where several packets can be in the buffer concurrently. Next, packets are transmitted sequentially out of the FIFO buffer in correct order. When the filtering is completed after the first word of every packet, the module induces only the inevitable delay of a single cycle.

B. Parsing

The parsing of the protocol stack is done in parallel while writing the input to the data buffer. This only interferes with the data flow if a header boundary lies within a word. In that case, the flow is kept for an additional cycle. The process is implemented by a finite-state machine, in which each node corresponds to one layer in the stack. This mechanism is shown in Figure 2.

The first node takes a special role, because the link layer is restricted to Ethernet. The Ethernet frame header has a minimum length of 14 bytes and it can be extended by multiples of 4 bytes (e.g. by an additional VLAN tag). This does not affect the headers inside, which are mostly aligned to 32 bit. As a result, a lot of fields span across two data words. To simplify the extraction of header fields, we provide a virtual word stream aligned to the start of the network header.

After the link layer node has determined the length of the header, it stores the remainder of the current word into a buffer

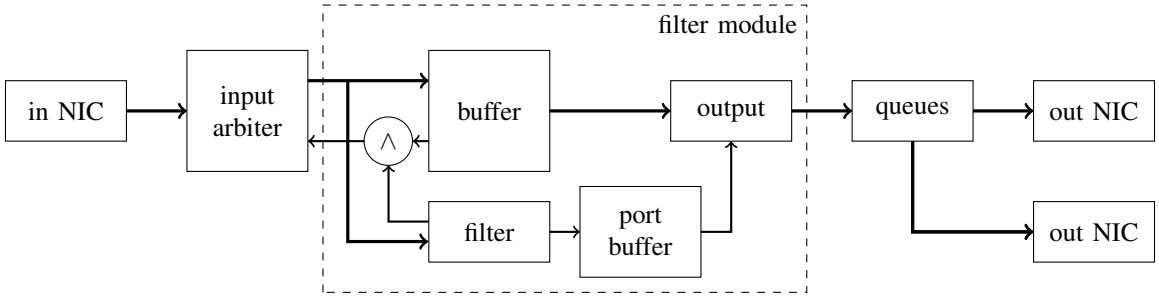

Fig. 1. Dataflow of packets through the modules of the NetFPGA and the feedback of the filter to stall

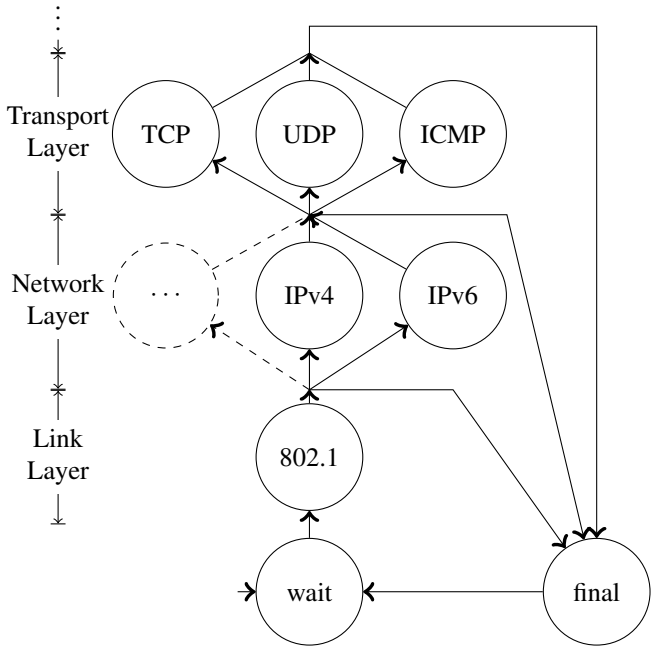

Fig. 2. Finite-state machine for parsing of the main internet stack

register. This register serves as the lower part of the virtual word. Inside the header, interesting fields can be extracted to apply a filter afterwards. One field that must be extracted to parse the network header is the ether type field, which indicates the protocol that is encapsulated by the payload.

Therefore, the structure of the consecutive nodes is characterised by a switch statement, selecting the protocol specified by the preceding layer. Each case must specify an offset to the next header and a type for the nested protocol. The positions of the required fields can easily be calculated within the virtual word, which is completed by the first part of the current AXIs word. Also a check has to be included, if the header spans multiple words to update the front part of the virtual word and increment the word counter to keep track of the position within the packet.

On completion of one node, the flag to retain the data flow is set and the state machine is transitioned to the state of the next layer. This is necessary to move the FSM into a consistent state before parsing the next protocol header. If the encapsulated header is not recognised, the transition goes directly to the final state of the FSM. In the final state, a check is performed to determine whether the packet is already completely received. This works by verifying that the current word is the last word of this packet. If the packet is completely received, the process uses an additional cycle in which the previously extracted header fields are stored to registers. If the packet is not completely received, the filtering actions described in the next section are performed in parallel to storing the input stream into the FIFO buffer. In the second cycle of the final state the extracted fields are matches against the specified filter.

C. Filtering

When the collection of header fields as part of the parsing process is finished and when all fields are stored in temporary registers, these can be matched independently of their former position in the packet. A rule in this filter process consists of a mask and a filter for each head field. Utilising the capabilities of the FPGA, all fields can be matched to multiple rules at once. These results are all conjoined by $<<$ or $>>$, so one matching rule suffices to mark the packet as a match.

The host can write and read the mask and filter data via the AXI bus over the PCIe interface. To facilitate the creation of the filter bit strings we adapted the python script from SCOTT. There the fields, parsed in hardware, are mapped to names. With this generator, human readable rules can be edited in a text file and then be compiled to native commands and directly written to the registers in the required format.

IV. EVALUATION

The intent of our work was to implement a linespeed filter on the NetFPGA. To verify this goal, we assembled a generic test setup. This consists of the NetFPGA card at its core and two additional systems, each equipped with an intel 10G X520-2 NIC. The topology between the three entities is a ring. This allows us to have a direct reference link for all measurements. As all ports have SFP+ connectors, we use direct attach copper cables.

A. Latency

For linespeed connections, latency is the most crucial factor. Because the test setup is laid out symmetrically, the delay can be measured by the round trip time (RTT) and precise time synchronisation of multiple systems is not required. The

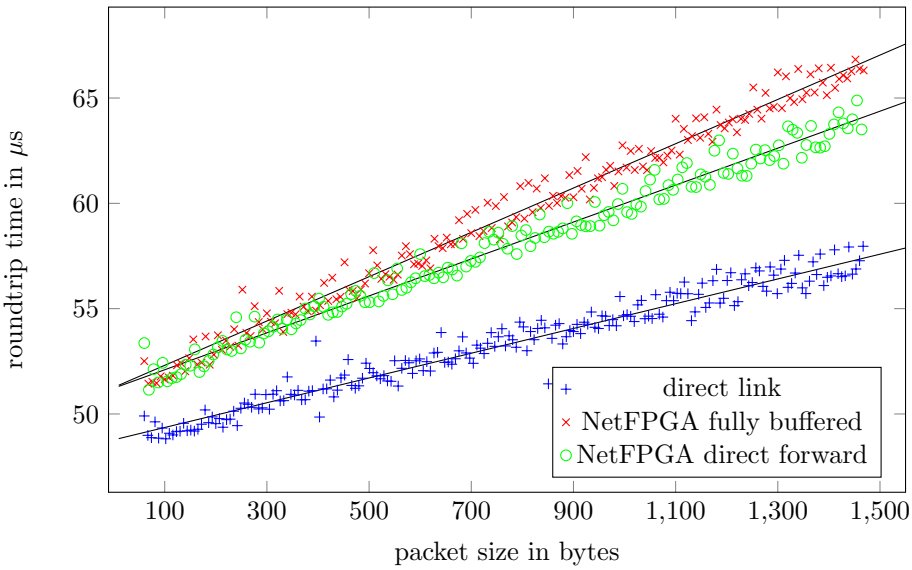

Fig. 3. Roundtrip time of ping echo request / -response depending on the packet size and route through test setup.

default utility `ping` was not adapted since the age of 10 Mbit Ethernet and is not accurate enough to measure these latencies. A user-space program in interrupt mode can not measure RTTs below ca. 180 μs. In super-user mode and adaptive send rate, the kernel switches to polling the I/O and thereby reducing the processing time on the host. With this method RTTs below 50 μs can be measured. To reveal minor deviations microsecond resolution is not sufficient.

To keep the ICMP protocol we adapted the ping utility to use the `timespec` struct and the corresponding functions of the realtime clock of linux instead of `timeval`, which enables nanosecond resolution. With this measurement, we achieve a reproducible standard deviation below 1.5 μs.

The duration of transmission of a packet from a memory location to the physical medium is dependent on the size of the packet. We have analysed this delay using the program `ping`, which can expand an ICMP packet with random data to a given size. Figure 3 presents this dependency for the two possible routes ant two configurations of the NetFPGA. We assumed a worst-case scenario, in which the packet is not forwarded until it is completely stored in the buffer. This corresponds to the use case where the destination port of the packet can not be determined until the last word of the packet.

The experimental results fully comply with our prediction that larger packets need more time. In case of the direct link, there is only one transmission for each direction, and it has the lowest RTT. The gently inclining slope in Figure 3 is caused by the buffer in the Ethernet cards. With an additional hop in the path, the inevitable delay can be seen as offset between the linear regressions at size zero. As the size includes headers, we cannot test this with an arbitrarily small packet size. In the buffered case, the time required for a packet of size zero would be the same as with direct forwarding. However, the additional buffer steepens the slope. The delay difference for a 1500 byte packet between buffered and direct of 2.6 μs coincide in order of magnitude with the theoretically calculated value of 480 ns.

B. Throughput

The second parameter of data transfer is throughput. We use `iperf` for this measurement, which sends random data over a TCP connection. This method is suitable for our case, because it resembles a common traffic of the Internet. To handle up to 10 Gb/s by the sending and receiving nodes, some kernel parameters have to be increased, such as the buffer sizes. Similarly, TCP requires an increased window size. To determine values for these parameters that are sufficient to fully utilise the connection, a direct link is used.

With jumbo frames sent over the direct link between the NICs, a throughput of about 9.9 Gb/s could be reached, but the data path of the NetFPGA is limited to the maximum transmission unit (MTU) of 1500 bytes. This reduces throughput to 9.5 Gb/s. By controlling the MTU on the link, the packet size can be adjusted. Figure 4 shows the throughput for the three different settings of the latency test.

With smaller packet sizes, the ratio of header size to payload length gets bigger and consequently, the data throughput drops. The measurement on the direct link is in accordance with the theoretically derived exponential decline. The throughput of packets with sizes below 1000 bytes through the NetFPGA drops below that measured on the direct link. This might be caused by multiple buffer registers placed into the data flow.

At common MTU sizes in the range from 1000 to 1500 bytes, the filter can keep up with the throughput over the direct connection. This only holds if the MTU is a multiple of 4 plus 2. As the MTU does not include the Ethernet header of 14 bytes, the peaks are at multiples of 4 bytes. A simple explanation is a bus of width 4 somewhere on the card. This only partially holds, because to transfer a 1400 byte packet over a 4 byte bus, 350 words are required. So the ratio of

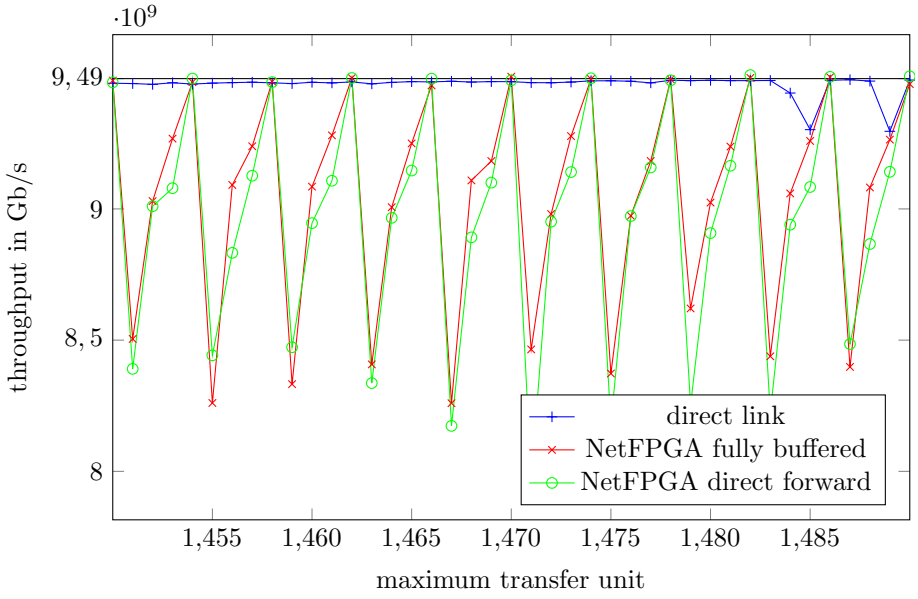

Fig. 4. Throughput measured with `iperf` depending on the the maximum transfer unit

$\frac{350}{351}$ is too small in relation to the drop from 9.5 to below 9 Gb/s. Given this, we suspect that the four lanes of the XAUI connection the PHY device to the FPGA, which may have huge overhead for unaligned word lengths.

However if the packet size is appropriate, the filter has no influence on the throughput of the intersected link.

V. Conclusion

Our protocol parser and filter for the NetFPGA architecture fulfils all our initial requirements of a linespeed network appliance as part of the NetFPGA environment which is easy to adapt. We implemented a self-contained intellectual property core ready to place in other NetFPGA projects as a preprocessing step.

Our parser uses an extended finite-state machine to parse byte-oriented protocol headers on multiple layers. The provided environment allows an easy adaption as well as the direct implementation of new protocols. This enables short development cycles in both the design and validation of these implementations. The processing rate of one header per cycle is comparable with dedicated 10 G network hardware. Our measurements confirm that the parsing and filtering process has no influence on throughput at adequate packet sizes, and a negligible influence on throughput for small packets, which do not carry payload but only handle latency dependant controls (TCP ACK). With respect to latency, the worst-case scenario imposes about 3 μs additional delay, plus the unavoidable delay of adding an additional hop on the medium access layer.

With standardised interfaces, our work can be incorporated in any project requiring filtering for a specific protocol, including the intrusion detection mechanisms we discussed as part of the related work.

For future work, the module become even more dynamic, because the synthesis of hardware is a complex time-consuming process. With one cycle read lookup-tables in on-chip block-RAM, it will be possible to specify the structure of the headers from the host at runtime. The expressiveness of the filter rules would be another area of interest. For example, the possibility to specify negative rules would provide all the expressiveness of predicate logic.

References

[1] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment 4: Media Access Control Parameters, Physical Layers, and Management Parameters for 40 Gb/s and 100 Gb/s Operation," IEEE Std 802.3ba-2010 (Amendment to IEEE Standard 802.3-2008), IEEE, pp. 1–457, 2010.

[2] B. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002. Proceedings*, 2002, pp. 111–120.

[3] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 99–107, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1055626.1055640

[4] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10Gbps FPGA-Based network intrusion detection system," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, P. Y. K. Cheung and G. A. Constantinides, Eds. Springer Berlin Heidelberg, Jan. 2003, no. 2778, pp. 880–889. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-45234-8_85

[5] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ser. Lecture Notes in Computer Science, M. Glesner, P. Zipf, and M. Renovell, Eds. Springer Berlin Heidelberg, Jan. 2002, no. 2438, pp. 404–413. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-46117-5_43

[6] M. Scott, "A wire-speed packet classification and capture module for netfpga," *First European NetFPGA Developers' Workshop*, 2010.

[7] P. Kobiersky, J. Korenek, and L. Polcák, "Packet header analysis and field extraction for multigigabit networks," in *Design and Diagnostics of Electronic Circuits & Systems, 2009. DDECS'09. 12th International Symposium on*. IEEE, 2009, pp. 96–101.