Pigasus: Efficient Handling of Input-Dependent Streaming on FPGAs

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Zhipeng Zhao

B.S., Electrical Engineering, Beihang University

M.S., Electrical Engineering, Beihang University

Carnegie Mellon University

Pittsburgh, PA

August 2021

Acknowledgments

First and foremost, I would like to thank Prof. James C. Hoe, who is a great mentor and friend. He taught me how to think critically and present clearly, and offered me a comfortable environment to grow as a researcher. As a 'general-purpose' advisor, James gave me so much invaluable advice, which helped me get through the difficult times of my PhD and helped me better know and improve myself. Thank you, James; without your guidance, support, and encouragement, I would not have had such a happy and enriching PhD journey.

I want to thank my committee members, Prof. Justine Sherry, Prof. Vyas Sekar, and Dr. Eriko Nurvitadhi, for the constant help and feedback on my thesis and project. It is my great fortune to have the opportunities to collaborate with Justine and Vyas, who taught me not only the basics of networking and systems, but also writing and speaking skills. Special thanks to Justine for inviting me to join her group activities, especially the daily check-ins since the beginning of the COVID-19 pandemic, which were tremendously useful for keeping myself productive and staying in good mental health while working from home for more than one year. I thank Vyas for his constructive feedback that helped me to extract the key ideas of my project and articulate an attractive story. I thank Eriko for providing the industry perspective and great collaboration opportunities that inspired my research and broadened my vision. I thank Dr. Michael Papamichael and Dr. Adrian Caulfield for mentoring me during my internship with the Catapult team at Microsoft, which

motivated me to explore the intersection of FPGA and Networking. I thank Prof. Jun Wang at Beihang University for introducing me to FPGA and encouraging me to pursue a PhD abroad.

I thank my collaborators. I thank Hugo Sadok and Nirav Atre for their contributions to Pigasus 1.0. Special thanks to Hugo for helping me remotely set up and maintain the clusters at my apartment since the beginning of the pandemic. I thank Joe Melber, Siddharth Sahay, and Shashank Obla for their contributions to Pigasus 2.0. I would also like to acknowledge my other collaborators, Mohamed Ibrahim, Andrew Boutros, Dr. Tommy Tracy, Prof. Kevin Skadron, Dr. Daniel S. Berger, and Prof. Aurojit Panda.

I also want to thank my other colleagues at CMU. I thank A-levelers: Michael Papamichael, Gabriel Weisz, Yu Wang, Marie Nguyen, Guanglin Xu, Prof. Tze Meng Low, Doru Thom Popovici, Guo Qi, Jing Huang, Jiyuan Zhang, John Filleau, Anuva Kulkarni, and Milda Zizyte. I received tremendous help from them, and I really miss those days we worked together and hung out for lunch. I feel very lucky to have had the chance to interact with many fellows in SNAP LAB during the network seminar and reading group, and to receive valuable feedback from them, especially Rukshani Athapathu, Christopher Canel, Val Choung, Rohan Gandhi, Anuj Kalia, Daehyeok Kim, Alan Liu, Grace Liu, Antonis Manousis, Adithya Abraham Philip, and Ranysha Ware.

Beyond CMU, I am very fortune to have many friends who've kept me company over the years. I want to thank my roommates Zhou Fang, Abby He, and Qi Yan for their help since my first day in Pittsburgh and for the fun times we have experienced together. I thank Xuetong Zhai, Xiaoying Sun, and Ying Liu for making delicious food and sharing cooking skills with me during parties.

I thank my parents Xinsheng Zhao and Yulan Chang for their unconditional love and support throughout my life. Without them, I would never have enjoyed

so many wonderful experiences. I thank our cat Ben Ben, who disliked my petting but still brought me tons of happiness. Lastly, I am truly grateful to my significant other, Liuyan Hu, for her constant support and encouragement. I love you.

This work is supported in part by SRC/JUMP (CONIX), NSF/VMware Partnership on SDI-CSCS, Intel/VMware Crossroads 3D-FPGA Academic Research Center and hardware donations from Intel.

Abstract

Field-programmable gate arrays (FPGAs) have achieved well-demonstrated success in a variety of real-time streaming applications, such as signal processing, machine learning inference, and video processing. In these applications, the processing accelerated by FPGAs is regular and '*input-independent*,' and thus the designs' behavior and performance are fixed. Accelerating real-time '*input-dependent*' streaming applications on FPGAs presents many interesting new challenges. For example, network intrusion detection and prevention systems (IDS/IPS) need to identify malicious network traffic, meaning different traffic will trigger different operations – thus activating different resource and performance bottlenecks – and making a static, fixed-performance FPGA design infeasible. Specifically, to achieve fixed performance, the design must allocate resources for handling worst-case scenarios, even if they happen rarely, thus losing the opportunity to use the same resources to improve common-case performance. For instance, since regular expression patterns are rarely triggered, the limited on-chip SRAM space could instead be used to make the string pattern matching component – which is exercised by almost every packet – larger and faster.

This thesis investigates novel design solutions to enable efficient handling of stream processing with input dependence in the problem context of IDS/IPS. The first part of this thesis focuses on achieving high performance under resource constraints for given inputs, by accelerating common cases while handling uncommon

cases efficiently at different levels of the system. Specifically, we propose three ideas: (1) FPGA-first architecture to allow the common datapath to sit entirely on FPGA fabric, while only offloading the complex, rarely triggered last pipeline stage to the CPU; (2) a fast-slow path design for TCP reassembly, where the rarely-triggered slow path can have a memory-saving data structure with non-deterministic performance without interfering with the performance of the fast path; and (3) hierarchical filters that use compact filters in front to keep up with the line-rate and thus reduce the resource consumption of the later, more expensive stages, which only need to keep up with the hit rate of the previous filter.

A key concern of the aforementioned design approach is that it is vulnerable to overfitting if the workload changes. The second part of this thesis tackles this problem, allowing for efficient and easy adaptation of the design to changing inputs at both compile time and runtime. In particular, we introduce two techniques: (1) a disaggregated architecture that enables easy scaling up, down, or out of particular components at *compile time* to cater to different expected traffic profiles; (2) a dynamic spillover mechanism to route the spillover traffic to backup streaming kernels that can be brought up on demand to absorb the increase in workload at *runtime*.

Pigasus, the 100Gbps IPS embodying the ideas in this thesis, has been open-sourced on https://github.com/cmu-snap/pigasus. End-to-end benchmarking with a variety of traces shows that Pigasus IPS can operate at 100Gbps using just 1 Intel Stratix 10 MX FPGA and an average of 5 cores of an Intel i9 processor, 50× more efficient than fixed-performance designs. The disaggregated architecture shows better scalability, reusability, and portability, with negligible performance and resource overhead relative to the static design. Finally, the dynamic spillover mechanism can prevent the performance degradation or resource wastage caused by the mismatch between the compile-time prediction of the traffic profile and the runtime real traffic profile.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, we have observed many successful instances where filed-programmable gate arrays (FPGAs) were used to accelerate real-time streaming applications such as signal processing [4, 5], machine learning inference [6, 7], video processing [8, 9], *etc.* In order to meet the high bandwidth and low latency requirements, developers typically need to extract the streaming processing kernels from applications and spatially map them onto FPGAs in a pipeline fashion. A beneficial characteristic of the above applications is that the processing accelerated by FPGAs is *input-independent*, meaning that each of the input units (*e.g.*, images) is processed by the same set of streaming kernels in the same order, and often with the same performance. As a result, the FPGA acceleration for these input-independent streaming applications has fixed behavior and performance.

Accelerating real-time, *input-dependent* streaming applications on FPGAs presents many interesting new challenges. In this thesis, we investigate how to efficiently handle input-dependent streaming using network intrusion detection and prevention systems (IDS/IPS) as a driving example. IDS/IPS are widely deployed

to identify and prevent network threats by scanning network packets (including payload) against a large number of rules [3, 10, 11, 12, 13, 14]. By nature, IDS/IPS performance is input-dependent, as different packets will utilize different resources and exercise different performance bottlenecks, thus making traditional static and fixed-performance streaming FPGA design strategies infeasible. To achieve high performance within the resource constraints, this thesis proposes a common-case aware and adaptive design approach that efficiently handles the input-dependent behaviors in a variety of deployment scenarios.

1.1 Motivation and Challenges

IDS/IPS are among the most performance intensive and complicated stateful network applications. Today, we are faced with the need to build IDS/IPS applications that can support line rates on the order of *100Gbps* [15] with *hundreds of thousands* [16] of concurrent flows[1] and can match packets against *tens of thousands of rules* [17][2].

Given the demanding performance requirements and complicated operations, today's software IDS/IPS are struggling to keep up. As shown in Figure 1.1, the state-of-the-art software IDS/IPS Snort 3.0 – which is equipped with the high-performance SIMD-accelerated Hyperscan [2] pattern-matching software library – can only achieve about 1Gbps per high-end CPU core, which is two orders of magnitude smaller than the 100Gbps line rate. Simply partitioning the traffic by flow and distributing the traffic across hundreds of cores is not a cost-effective option.

[1] A term used in the network community. Here, we define it as a 5-tuple, *i.e.*, source IP, destination IP, source port, destination port, protocol.

[2] There exist other models of IDS/IPS systems, including 'anomaly-based' models, which detect whether a system is operating normally based on heuristics, and 'script-based' models, such as Zeek [10, 11], which execute arbitrary user code over scanned traffic. These models are outside the scope of this work.

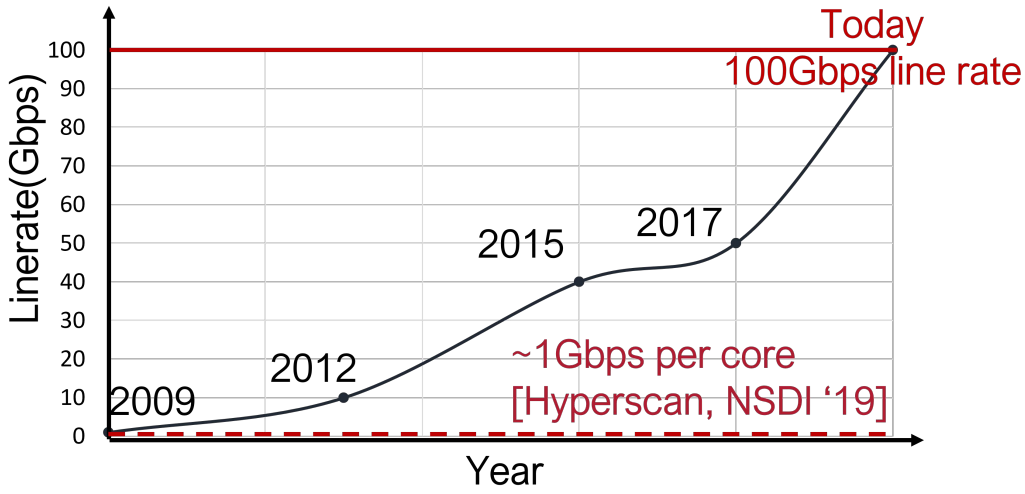


Figure 1.1: Line rate evolution [1] and state-of-the-art IDS/IPS performance [2]

This motivates us to pursue opportunities to accelerate IDS/IPS using FPGAs. Our goal is to support at least 100K flows and check at least 10K rules at 100Gbps line rates, using only *a single server* (explained more thoroughly in Section 2.1).

While many prior works have integrated FPGAs with IDS/IPS processing [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29], for the most part, these have followed traditional static and fixed-performance FPGA designs. These prior designs were designed for the worst-case scenario instead of the common case, making them excessively inefficient and prohibitively expensive. The high cost also restricts prior works to focus on accelerating particular sub-task of IDS/IPS, as it is considered too resource-intensive to try to map a complete IDS/IPS pipeline onto an FPGA. Unfortunately, as we will show in Section 2.4.1, this single-task fixed-performance offloading is insufficient to close the order-of-magnitude performance gap to offer 100Gbps end-to-end performance. Therefore, we need a new way to

efficiently handle input-dependent streaming on FPGAs.

1.2 Handling Input-Dependent Behaviors

This thesis presents new design techniques to efficiently handle input-dependent streaming using IDS/IPS as the main example. The key insights are (1) that we should fine-tune the design for common cases and handle uncommon cases efficiently at different levels of the system to extract maximum performance with minimal resources, and (2) that we should adapt the design for changing inputs at both compile time and runtime to address the brittleness of designs tailored to common cases.

To demonstrate the effectiveness of our method, we developed an end-to-end IDS/IPS, called *Pigasus*, which can achieve 100Gbps supporting 100K flows and 10K rules using a single server with one FPGA-based SmartNIC (Intel Stratix 10 MX 2100 FPGA) and, on average, 5 cores (Intel i9-9960X CPU). Pigasus is 100× faster than a CPU-only baseline and 50× faster than prior FPGA designs. Furthermore, it can be easily adapted to new deployment environments and shifting workloads. In the remainder of this section, we give an overview of the five key ideas behind Pigasus.

FPGA-first architecture: Traditional static and fixed-performance FPGA IDS/IPS accelerators [30, 29, 19] are excessively inefficient due to worst-case over-provisioning, such that it is not affordable to offload a complete IDS/IPS from CPU onto FPGA. We argue that in order to achieve a two-orders-of-magnitude speedup of IDS/IPS, we need to reverse the roles of FPGAs and CPUs. In an FPGA-first architecture, the FPGA serves as the primary processing unit, which has direct connections to 100Gbps network transceivers and performs most of the packet pro-

cessing. Specifically, the 'innocent' packets (common case) will be entirely processed by the highly parallel hardware datapath on FPGAs for high throughput and low latency. Only 'suspicious' packets (uncommon case) will ever reach the CPU, which works as the complexity offloader to the FPGA. This FPGA-first architecture makes it possible to reach 100× speedup with minimal resource consumption.

Fast-slow path SRAM-based TCP reassembly: Prior approaches to FPGA-based TCP reassembly use fixed-length (*e.g.* 64KB), statically allocated buffers implemented in DRAM [29, 28]. However, as the network speed evolves from 1Gbps or 10Gbps to 100Gbps and beyond, we need to use faster but much smaller FPGA on-chip SRAM to buffer packets and track their order. Our idea is to create a *fast path* for processing the in-order packets (common case) and a *slow path* for processing the out-of-order (OOO) packets (uncommon case). This allows us to use a memory-dense data structure to track OOO packets with little impact to overall performance.

Hierarchical pattern matching: Traditional input-independent streaming string search algorithms [19, 2] failed to scale to 10K rules while running at 100Gbps. Our solution is to separate the need of line-rate support and hit-rate support through hierarchical filters. The front-end compact filters keep up with the line-rate, checking all packets against all rules (common case), while the back-end expensive filters only need to keep up with the hit rate of the previous filters by checking suspicious packets against partially matched rules (uncommon case).

Disaggregated service-oriented streaming design: For many years, FPGA development has adhered to 'ASIC-style' design methodology; that is, once deployed, the design rarely changes. This is fundamentally unsuitable for handling input-dependent behaviors. To address this problem, we introduce a new design

methodology. By disaggregating the system, parameterizing individual streaming services, and connecting them to a common communication abstraction, the new methodology offers a selection of designs – rather than a single design – from which users can easily select the most efficient design at compile time as the deployment environment changes.

Dynamic spillover mechanism: Conventional fixed-performance FPGA designs are either impractical or not efficient enough to handle input traffic dynamism. In response, we propose a dynamic spillover mechanism that introduces more computing power on demand at runtime to absorb the temporary increases in workload. When bursty traffic shows up, the backup streaming kernels (*e.g.*, those implemented on CPU) are temporarily brought up to process the spillover traffic. For most of the time, the system does not need to allocate resources for these backup streaming kernels.

1.3 Thesis Contributions

This thesis investigates how to efficiently handle *input-dependent* streaming in the context of IDS/IPS on FPGAs. Specifically, it makes the following contributions:

- We propose an FPGA-first architecture, which inverts the conventional roles of FPGAs and CPUs, allowing a common datapath to sit entirely on FPGA and thus enabling a speedup of two orders of magnitude.
- We develop a fast-slow path SRAM-based TCP reassembly. By processing in-order packets on a fast path and out-of-order packets on a slow path, we can adopt a compact data structure on the slow path to save precious FPGA on-chip memory without disrupting the overall performance.

- We develop hierarchical pattern matchers that separate the need for line-rate support and hit-rate support, allowing for fewer replicas of expensive filters in later stages, thus saving resources without affecting the overall throughput.
- We introduce a disaggregated service-oriented streaming design methodology that addresses the brittleness of our common-case-tailored design as the deployment environment changes at compile time.
- We design a dynamic spillover mechanism, which can dynamically handle the case when expensive operations are triggered very frequently within a short period of time, by routing the spillover traffic to backup streaming kernels. This improves the resilience of our common-case-tailored design at runtime.
- To demonstrate the effectiveness of the above ideas, we develop an end-to-end FPGA-based IDS/IPS, called *Pigasus*. Pigasus successfully met our target of 100Gbps, 10K rules, and 100K flows on a single server. Pigasus is open-sourced at https://github.com/cmu-snap/pigasus and has been actively used by researchers.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the background of IDS/IPS and FPGAs and explains why the conventional wisdom does not work for input-dependent streaming. Chapter 3 presents the Pigasus system overview, which provides necessary context for understanding the five key ideas in this thesis. The first part of the thesis, including Chapters 4, 5, and 6, focuses on how to extract maximum performance for given inputs using minimal resources. Specifically, Chapter 4 presents the FPGA-first architecture. Chapter 5 presents the fast-slow path SRAM-based TCP reassembly. Chapter 6 presents the hierarchi-

cal pattern matching. The second part of the thesis, including Chapters 7 and 8, focuses on addressing the brittleness of our common-case-tailored designs as deployment environments change and workloads shift. In particular, Chapter 7 presents the disaggregated service-oriented streaming design. Chapter 8 presents the dynamic spillover mechanism. Finally, Chapter 9 concludes the thesis and discusses future directions.

Chapter 2

Background

This chapter first presents the background of IDS/IPS (Section 2.1) and FPGAs (Section 2.2). Then, this chapter explains what the *input-dependent* behaviors are in a real-world application and why they are unfavorable for FPGAs (Section 2.3). Finally, this chapter describes the conventional wisdom (Section 2.4) – the straightforward ways to implement Pigasus – and why they do not work, motivating this thesis.

2.1 IDS/IPS Background

The key goal of a rule-based IDS/IPS is to identify when a network packet triggers any of the up to tens of thousands of rules. A given rule may specify one or several *patterns* as shown in Figure 2.1. Patterns come in the following categories:

- *(a) Header match*: a filter over the flow 5-tuple (*i.e.*, source IP, destination IP, source port, destination port, protocol);
- *(b) Flow state*: check if the connection is 'open,' 'established,' or 'to server';

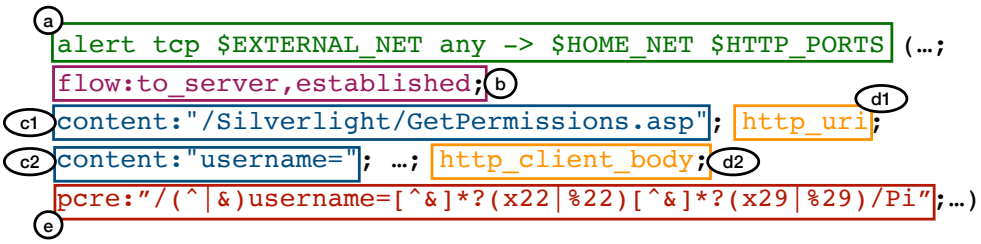


Figure 2.1: A simplified Snort-compatible rule

- *(c,d) String match*: an exact match string to be detected within the TCP bytestream or within a single UDP packet. One rule may contain multiple string patterns. In Snort, for performance reasons, one of the string patterns is selected as the special *fast pattern*. (d) refers to an identifier, specifying the region of the flow in which this string should be searched;
- *(e) Regular expression*: a regular expression to be detected within the TCP bytestream or within a single UDP packet.

Rules are detected at the granularity of a 'Protocol Data Unit' (PDU), *i.e.*, a signature is only triggered if all matches are found within the same PDU (not over the course of the entire flow). By default, a PDU consists of one packet, but it is possible to define other protocol-specific PDUs spanning multiple packets (*e.g.*, one HTTP GET request).

When an IDS/IPS operates in *detection* mode, a triggered rule results in an alert or an event to be recorded to a log. When an IDS/IPS operates in *prevention* mode, a triggered rule may raise alerts, record events, or *block* traffic from the offending flow or source. IPSes, hence, must operate inline over traffic and are latency-sensitive – *i.e.*, a packet may not be released to the network until after the IPS has completed scanning it. IDSes, on the other hand, may operate asynchronously and are often deployed over a secondary traffic 'tap' which provides

Figure 2.2: Simplified Snort [3] IDS/IPS block diagram

copies of the active traffic.

IDS/IPS components: Figure 2.2 illustrates the key components of Snort [3], the most widely-known open-source software IDS/IPS. In Snort, the network packet is first parsed to extract the metadata, including packet size, IP addresses, ports, *etc.* The Reassembly module is used to keep track of the TCP flow states (*e.g.*, sequence number) and reorder out-of-order (OOO) packets, whose sequence numbers are larger than the expected sequence numbers tracked by the flow table. The Multi-String Pattern Match (MSPM) module checks packets against *fast patterns* and header patterns. Specifically, the *fast patterns* of all the rules are constructed together, such that a packet can be checked against all of them in parallel in one pass. If the packet does not match any of the *fast patterns*, it will be safely identified as an innocent packet. Otherwise, it will be further checked by the Full Match stage, where the rest of the string patterns (*non-fast patterns*)[1] and regular expressions of the partially matched rules are fully evaluated. The Action unit conducts the alert, log, or block actions.

Input-dependent behaviors in IDS/IPS: As one may already notice, an IDS/IPS has *input-dependent* behaviors by nature. For instance, some innocent packets only need to be examined by the first three stages – Parser, Reassembly, and MSPM. In contrast, the suspicious packets – including malicious packets and packets that are falsely identified as positive – will be examined by all of the stages, thus taking

[1] In Pigasus, we move this portion out of the Full Match stage and implement it as another level of filter in MSPM on FPGA instead to improve the performance. We explain more details in Chapter 6.

more time. Such input-dependent behaviors exist in different aspects: the OOO packets take more time than in-order packets, one packet can trigger a variable number of partially matched rules each with variable performance, *etc.*

Software IDS/IPS performance: This work aims to be compatible with Snort rulesets. In our experiments, we primarily work with the Snort Registered Ruleset, which contains roughly 10,000 signatures [17]. This ruleset, combined with conversations with system administrators, sets our goal of supporting 10K rules. In addition, we target 100Gbps as the state-of-the-art line-rate [15] and we aim to support 100K flows. To the best of our knowledge, there exists no measurement study detailing how many flows to expect at 100Gbps, so we derive our 100K flow goal by extrapolating a two-orders-of-magnitude growth factor from a 2010 study [16].

In 2019, Intel published Hyperscan [2], an x86-optimized library for performing both string matching and regular expression matching. Hyperscan is the key new element in *Snort 3.0*, which is 8× faster than its predecessor [31, 32]. Nonetheless, we find that Snort 3.0 cannot meet our goal of supporting 100Gbps, 100K flows, and 10K rules on a single server.

We ran Snort 3.0 on a 3.6GHz server and measured the *single-core throughput* over 7 publicly available network traces (described more thoroughly in Section 3.3.1). We plot the results in Figure 2.3. This would require 125-667 cores or 4-21 servers to support 100Gbps of throughput, even with the generous assumption that Snort 3.0 is capable of perfect multicore scalability.

The performance (100Gbps line-rate, real-time action under IPS mode) and capacity (10K ruleset size, 100K flow states) requirements together with the complex operations introduced by the *input-dependent* behaviors make the IDS/IPS a very challenging application to accelerate.

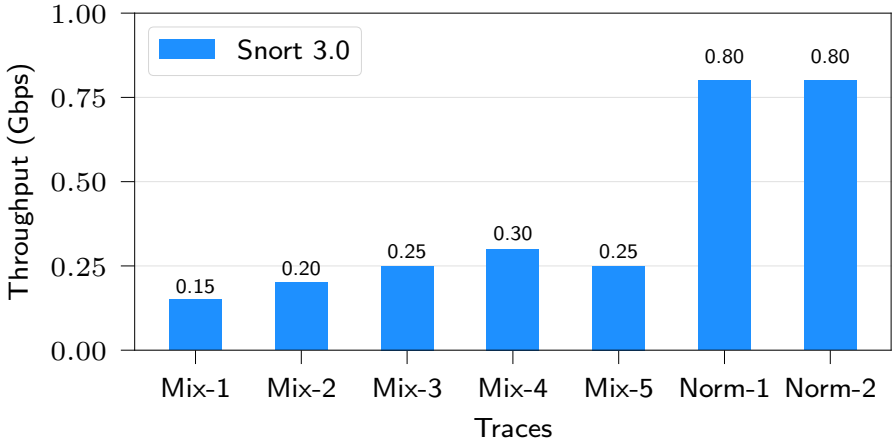


Figure 2.3: Single-core, zero-loss throughput achieved by Snort 3.0 with Hyperscan over a range of empirical traces

2.2 FPGA Background

Why look to FPGAs to improve IDS/IPS? While there are many platforms ('accelerators') that offer highly parallel processing, FPGAs are most promising because (a) they are conveniently deployed as SmartNICs[2] where they are poised to operate on traffic without PCIe latency or bandwidth overheads [1, 33, 34], (b) they are more flexible at interacting with memory and implementing customized data structures compared to P4-switches [35, 36], (c) they are energy-efficient (using 4-5× [37, 38, 39] fewer Watts than GPUs), and (d) they are more adaptive to various changing deployment environments than ASICs and are programmable to allow patches when new exploits emerge [40].

[2]Smart Network Interface Controller (SmartNIC) refers to an emerging type of NIC, which extends traditional NICs with advanced functionality using programmable devices such as embedded CPUs or FPGAs to accelerate the tasks that are normally processed by host CPUs.

FPGA compute: FPGAs allow programmers to specify custom circuits using register-transfer-level (RTL) code. However, implemented naïvely, FPGA-based designs can be much slower than their CPU counterparts because FPGA clock rates operate 5-20× slower than traditional processor clock rates. To achieve performance speedups relative to CPUs, circuits must be designed with a high degree of parallelism. FPGAs achieve parallelism either through *pipeline parallelism*, in which *different* modules operate simultaneously over different data, or through *data parallelism*, in which copies of the *same* module are cloned to operate simultaneously over different data.

FPGA memory: Today's FPGAs offer programmers a suite of memory options. Block RAM (BRAM) is the primary FPGA on-chip SRAM, because read requests receive a response within one cycle. Furthermore, BRAM is very friendly to parallelism. Our target FPGA (Intel Stratix 10 MX FPGA Development card [41]) offers 16MB of BRAM distributed to 6847 20Kb *blocks*. Divided into 20Kb blocks with two ports each, it is possible to read from all BRAM blocks in parallel (and each BRAM block twice) per cycle. When a developer wishes to issue more than two parallel reads to a BRAM block per cycle, they may choose to *replicate* the block to allow more simultaneous access to stored data.

Our target FPGA also offers 8GB of on-board DRAM (which takes about 100 nsec or 20 cycles of latency when running at 200MHz between read request and response) and 10MB of embedded SRAM (eSRAM), another type of FPGA on-chip SRAM (which takes fixed 12 cycles of latency between read request and response). Because of the multi-cycle latency for these two classes of memory, they are not suitable for storing data that must be read/written every cycle. Furthermore, both are more bandwidth-limited than BRAM and offer fewer parallel lookups. However,

as we will discuss in Section 3.2, pushing what data are feasible into these slower classes of memory is necessary to free up as much BRAM as possible to support fast common-case memory-intensive processing.

2.3 Why Input-Dependent Behaviors Are Challenging for FPGAs?

If an application is *input-independent*, each input unit (*e.g.*, a packet or an image) will be processed by the same set of logic with fixed performance, yielding a static and relatively simple design. Most prior FPGA accelerators fall into this category, which also explains why FPGA design style has been static, *i.e.*, assuming a one-shot design will work for all scenarios.

However, if an application is *input-dependent*, FPGAs have to react differently based on the content of the input, leading to a much more complicated, dynamic, and non-deterministic design, requiring a different design mindset. In the context of IDS/IPS, FPGAs have to deal with innocent traffic and malicious traffic differently and handle a variable number of matches and all kinds of corner cases, all of which have to be realized under strict performance and resource constraints. Since the input could change drastically, any one-shot design will either cause performance degradation or resource wastage as the deployment condition changes.

Simply adopting the conventional static, fixed-performance designs, which requires provisioning for the worst case, is too inefficient and expensive to be practical. Therefore, we need a design that can achieve high performance using minimal resources and be adaptive enough for shifting workloads and changing environments.

2.4 Conventional Wisdom

In this section, we briefly discuss the conventional ways to implement IDS/IPS and why they do not work.

2.4.1 Traditional FPGA Offload Paradigm

To avoid the challenges introduced by input-dependent behaviors, the traditional FPGA offload paradigm favors designing for the *worst case*. Designing for the worst case makes the FPGA implementation simple, regular, and deterministic, but would incur resource over-provision. For example, Grapefruit [42], a state-of-the-art regular expression engine for FPGAs, statically maps all regular expression patterns on FPGAs to allow *input-independent* performance – matching one regular expression takes the same time as matching all regular expressions. Unfortunately, this is overkill, as we rarely need to check all regular expressions in the Full Matcher, but the unused patterns still consume valuable FPGA resources.

Since designing for the worst case is so expensive, prior FPGA-accelerated IDS/IPS work can only afford to accelerate a single task of the IDS/IPS end-to-end system. For example, in many works [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30], FPGA is the 'offloader' of a *specific task* and the CPU is the 'primary' processing unit, performing the majority of the processing and dealing with the input-dependent behaviors. However, as we will show in Section 4, after the Hyperscan improvement of the MSPM task by 8×, there is no single dominating task in Snort 3.0. Just accelerating a specific task on FPGA will not give us orders of magnitude end-to-end speedup based on Amdahl's law.

2.4.2 Fixed-Buffer-Size TCP Reassembly

Hardware design often favors data structures that are *fixed-length*, *constant-time*, and generally *deterministic*, and most TCP reassembly designs follow suit. For instance, Yuan *et al.* [29] allocate a fixed 64KB packet buffer in DRAM and use 7 pairs of pointers to track the OOO state for each flow. By using static buffers, these designs can achieve nearly constant-time insertion of out-of-order packets into memory; furthermore, the memory consumed by any individual flow is fixed, so freeing space is also deterministic. However, as the line-rate is increasing rapidly, from 1Gbps or 10Gbps to 100Gbps and beyond, DRAM speeds are struggling to keep up.

While using fast FPGA on-chip SRAM will address the speed problem, it presents new challenges – the on-chip SRAM capacity is many orders of magnitude smaller than DRAM capacity. Simply adopting the statically allocated fixed-size buffer approach does not work for SRAM. For example, allocating 64KB for *every OOO flow* [28] would require 64MB to support 1K OOO flows (typically 1% of the 100K flows).

2.4.3 Input-Independent Multi-String Pattern Matching

Classic FPGA-based multi-string pattern matching designs adopt state-machine approaches, which have input-independent performance [19]. In these designs, the multiple string patterns are constructed as a state machine; each byte of the input stream invokes a state transition. A rule will be matched when the input bytestream exercises a certain state transition path, reaching the final state. The performance is *input-independent*, since the system can always accept a fixed number of bytes per cycle, no matter how many rules these bytes trigger. However, state-machine-based

designs are very memory- and/or logic-consuming, and hence only work for up to *a few hundreds of rules*.

An alternative is to use the Hash Table-based approach as used in the Hyperscan software library [2]. In this approach, the string patterns are hashed to construct a Hash Table at compile time. During runtime, the hashed values of the input bytes are used to index the Hash Table, determining whether or not there is a match. When porting this algorithm from software to FPGA, the conventional wisdom of input-independent acceleration would require many replicas of the matcher to keep up with the line-rate, exhausting the on-chip memory resource again.

2.4.4 ASIC-Style Design Methodology

For a long time, FPGA development (including *Pigasus 1.0*, the initial implementation of Pigasus) has been following the ASIC-style design methodology [43] – once a design is deployed it rarely changes, just like an application-specific integrated circuit (ASIC). This design style is fundamentally mismatched from the input-dependent behaviors in applications.

As we have discussed, the key to extracting maximum performance using minimal resources is to tune for the common case. However, as the deployment environment changes, the common/uncommon case ratios could be different from the initial setting, requiring re-tuning of the design to retain high efficiency, like scaling out the performance-bottleneck sub-components for the new workload. With the ASIC-style design methodology, such re-tuning requires digging into the codebase and manually modifying the RTL code, which is challenging and unproductive.

2.4.5 Static Handling of Burstiness

Since the design decision made at compile time may not accurately predict the traffic profile at runtime, it is possible that the slow operations are triggered unexpectedly frequently in a short period of time, leading to performance degradation. The traditional FPGA designs either allocate enough processing power for the worst case or introduce a big enough buffer to absorb the burstiness. Unfortunately, as we discussed in Section 2.4.1, the worst-case-oriented designs are impractical, as the resources needed are far beyond FPGA capacity.

Big buffers also do not work as (1) any small jitter in the 100Gbps traffic can easily overflow our MB-level of buffer, and (2) we cannot buffer indefinitely due to latency concerns.

Chapter 3

Pigasus Overview

In this chapter, we present our research artifact, which provides necessary context for understanding the key ideas of the remaining chapters. We first give an overview of Pigasus and then describe our memory choices at system level before evaluating the end-to-end system performance. In the next chapters, we will dive deeply into each individual idea behind Pigasus.

3.1 Pigasus Datapath

Figure 3.1 depicts the major components of Pigasus' architecture. The Parser, Reassembler, and Multi-String Pattern Matcher (MSPM) are implemented in the FPGA while the Full Matcher (the non-fast string pattern matching portion is implemented in MSPM on FPGA, as explained in Chapter 6) is offloaded to the CPU. The following describes how these pieces work together.

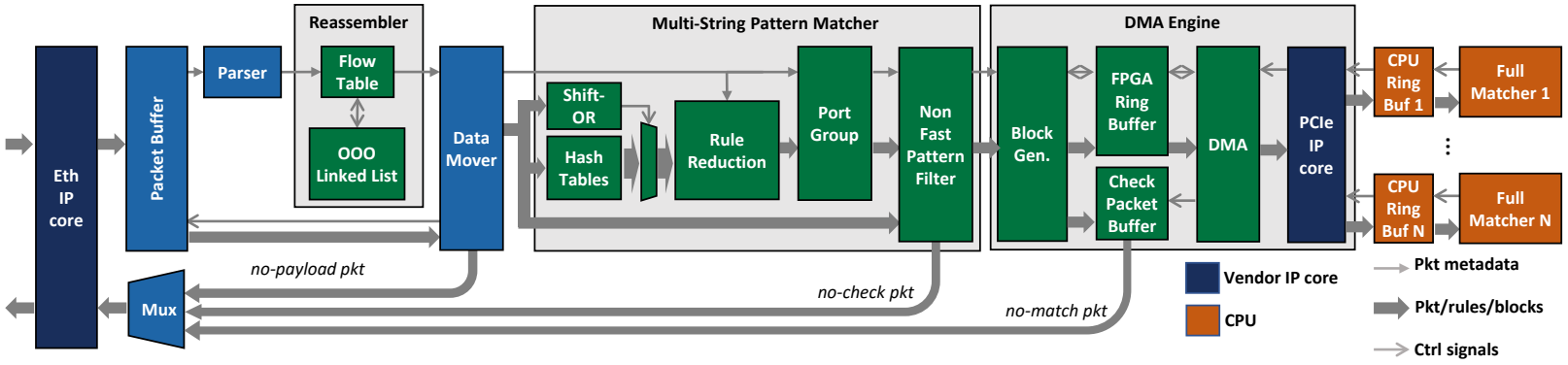


Figure 3.1: Pigasus' architecture

Initial packet processing: Each packet first goes through a 100Gbps *Ethernet Core* that translates electric signals from an Ethernet cable into raw Ethernet frames. These frames are temporarily stored in the *Packet Buffer*; each frame's header is separately sent to the *Parser* – which extracts TCP/IP header fields as metadata (*e.g.*, sequence numbers, ports) for use by the Reassembler and MSPM – and then forwards the header to the Reassembler.

Reassembler: The Reassembler sorts TCP packets from the same flow in order so that they can be scanned contiguously (*i.e.*, to identify matches that span across multiple packets). The Reassembler is able to record the last few bytes of the packet's predecessor in that flow in order to enable cross-packet search in the MSPM. UDP packets do not need reassembly and thus are forwarded through this stage without processing. The key challenge in designing the Reassembler is doing so *at line-rate* with state for *100K concurrent flows* (explained more thoroughly in Section 2.1).

Data mover: While the Parser and Reassembler operate on headers and metadata alone, the MSPM operates on full packet payloads. The Data Mover receives the (sorted) packet metadata from the Reassembler and issues requests to fetch raw packets from the *Packet Buffer* so that they can be forwarded to the MSPM. This module decouples the data movement task from Reassembler packet processing tasks to give the Reassembler more cycle budget on processing packets.

Multi-string pattern matcher: The MSPM is responsible for (a) checking every packet against the header match for all 10,000 rules (explained more thoroughly in Section 2.1), (b) *every index of every packet* against the *fast pattern* for all 10,000 rules, and (c) checking the *non-fast string patterns* of a partial matched rule (this functionality is in the Full Matcher in Snort, but Pigasus' design accelerates this

task on FPGA for better performance). Thus, it has the most demanding performance requirements among Pigasus modules. At a 400MHz clock rate, the payload matching alone requires searching 32 indices against all 10,000 signatures *every clock cycle*.

DMA engine: For each packet, the MSPM outputs the set of *rule IDs* that the packet partially matched. If the MSPM outputs the empty set, the packet is released to the network; otherwise, it is forwarded to the DMA Engine which transfers the packet to the CPU for Full Matching. To save on PCIe bandwidth, the DMA Engine keeps a copy of the packets sent to the Full Matcher in FPGA-side DRAM (*Check Packet Buffer*); this allows the Full Matcher to reply with a (packet ID, decision) tuple as a response, rather than copying the entire packet back over PCIe after processing.

Full matcher: On the software side, the Full Matcher polls a ring buffer which is populated by the DMA Engine. Each packet carries metadata including the *rule IDs* that the MSPM determined to be a partial match. For each rule ID, the Full Matcher retrieves the complete rule (regular expressions and other fields of the rules) and checks for a *full match*. It then writes its decision (forward or drop) to a transmission ring buffer, which is polled by the DMA Engine on the FPGA side. If the decision is to forward, the DMA Engine forwards the packet to the network; otherwise, the packet is simply erased from the DMA Engine's Check Packet Buffer.

3.2 Memory Resource Management

The core obstacle to seeing Pigasus fully realized is fitting all of the above functionality (except the Full Matcher) within the limited memory on the FPGA. As

discussed in Section 2.2, BRAM is the 'best' of the available memory: it is the only class of memory that can perform read operations in one cycle, and it is also the most parallel. However, it is limited to only 16MB even on a high-end Intel Stratix 10 MX FPGA. In this section, we discuss how we choose appropriate types of memory for different components to save BRAM, which is orthogonal with introducing new resource-efficient algorithms described in the following chapters (Chapter 4, Chapter 5, and Chapter 6).

In Pigasus, BRAM is reserved only for modules which require either latency-sensitive or throughput-demanding memory accesses, namely the Reassembler and the Multi-Pattern String Matcher. Specifically, the Reassembler performs flow table lookup and update for each packet and thus requires low-latency memory access to improve the packet rate. The MSPM requires high throughput as every index of every packet must be checked against multiple tables (*e.g.* ShiftOR table, Hash Table, different header tables, *etc.*) at 100Gbps.

To save BRAM, the other stateful modules such as the Packet Buffer and DMA Engine are allocated to less powerful eSRAM and DRAM respectively.[1] eSRAM and DRAM turn out to be sufficient for these tasks because the Packet Buffer and DMA Engine have much less stringent demands in terms of bandwidth and latency. In the case of the packet buffer, packet data is written and read only once and hence bandwidth demand is low but still exceeds DRAM's peak throughput; the data mover prefetches each packets 12 cycles in advance of pushing it to the MSPM keeping throughput high with a negligible latency impact. The DMA Engine uses DRAM – which has the highest and variable latency and the lowest bandwidth – for

[1]FPGA manufacturers have been experimenting with varied classes of memory on-board the FPGA over the past few years. From the manufacturers' perspective, Pigasus can be seen as a success story for how varied memory enables more diverse applications which tailor their memory usage to per-task and data structure demands.

the Check Packet Buffer. Since on average only 5% of packets require Full Matching functionality, this places little stress on DRAM bandwidth; the latency overhead of DRAM, while high when compared to BRAM, is still 10× faster than the PCIe latency suffered by packets sent to the CPU for full match.

Even though this leaves almost[2] the full capacity of BRAM for the Reassembler and Multi-String Pattern Matcher, realizing these modules is challenging. For instance, using traditional NFA-based search algorithms for the MSPM, given the public ruleset, would require 23MB – more than our 16MB BRAM capacity. Similarly, statically allocating 64KB of out-of-order buffer per flow for even 1K flows (out of 100K flows we need to support) easily exceeds 16MB. In the next three chapters (Chapter 4, Chapter 5, and Chapter 6), we will show how our novel designs efficiently use the limited BRAM without compromising performance.

3.3 Evaluation

In this section, we evaluate Pigasus and show that:

- Pigasus is at least an order of magnitude more efficient than state-of-art Snort running in software, using $23-200\times$ fewer cores and $18-62\times$ less power;
- Pigasus' performance gains are resilient to a variety of factors such as small packets and the rule-match profile of the traffic;
- The Pigasus architecture actually has resource headroom, suggesting a roadmap for handling even more complicated workloads.

We start by describing the evaluation setup we use for the rest of the section before the detailed results.

[2]Some internal buffers/queues do use BRAM.

3.3.1 Setup

Implementation: We implement Pigasus using an Intel Stratix 10 MX FPGA Development card [41] as the SmartNIC in a 16-core (Intel i9-9960X @ 3.1 GHz) host machine. The Stratix 10 MX FPGA has 16MB of on-chip BRAM, 10MB of eSRAM, and 8GB of off-chip DRAM. To implement Pigasus' CPU/software components, we adapt Snort 3.0 to allow it to receive reconstructed PDUs and rule IDs, coming from the FPGA directly into its Full Matcher. We run Snort 3.0 software experiments in an Intel i7-4790 CPU @ 3.60 GHz.

Traffic generator: We installed both DPDK Pktgen [44] and Moongen [45] on a separate 4-core (Intel i7-4790 @ 3.6 GHz) machine with a 100Gbps Mellanox ConnectX-5 EN network adapter. DPDK Pktgen achieves higher throughput when replaying PCAP traces – up to 90Gbps – and hence we use the DPDK Packet Generator when running experiments with recorded traces. Moongen is better at generating synthetic traffic at runtime and can do so at up to the full 100Gbps offered by the underlying network. We specify in each experiment which traffic generator was used.

Traces and ruleset: We test Snort and Pigasus both using the publicly available Snort Registered Ruleset (snapshot-29141) [17] and different traces from Stratosphere [46, 47]: *CTU-Mixed-Capture-1–5*, *CTU-Normal-12*, and *CTU-Normal-7*. We refer to them as *mix-1–5*, *norm-1*, and *norm-2*, respectively. For the *mixed* traces, we use the `*before.infection` pcaps. We use Stratosphere traces because their packet captures contain the original payloads, which is essential when evaluating IDS/IPS. However, such traces are not captured at 100Gbps[3]. To test the performance of Pigasus, we replay the trace at the full speed of the DPDK

[3]We could not find any public 100Gbps traces with meaningful payload.

Pktgen.

Measuring throughput and latency: We measure throughput in two ways: 1. The *Zero Loss* throughput is measured by gradually increasing the packet generator's transmission rate until the system (Snort or Pigasus) first starts dropping packets; 2. The *Average* throughput is computed as the ratio of the cumulative size of packets in the trace (in bits) to the total time required to process the trace. We measure latency (at low load) using DPDK Pktgen's built-in latency measurement routine. Unfortunately, DPDK embeds timestamps in the packet body, which never triggers the CPU-side Full Matching functionality. Instead, we measure the end-to-end latency for Pigasus on an empirical trace using FPGA-side counters, and then adding the baseline FPGA loopback latency to it.

3.3.2 End-to-End Performance and Costs

In this section, we compare the performance, power, and cost of Pigasus vs. Snort and provide the resource breakdown of Pigasus.

Provisioning for 100Gbps throughput: Figure 3.2 reports the number of server cores required to achieve 100Gbps for the evaluated Stratosphere traces for different settings. The top half is under the assumption of loss-free processing without buffering, while the bottom reports the steady-state core requirements based on the assumption that we have an infinitely big buffer to buffer packets during the peak periods and defer the full matching to allow the cores to catch up after the peak has passed.

The Pigasus results are based on experiments where the system is tested at increasing numbers of cores at maximum throughput, until we observe no packet loss. However, mix-1 is special as the hardware datapath is the bottleneck which

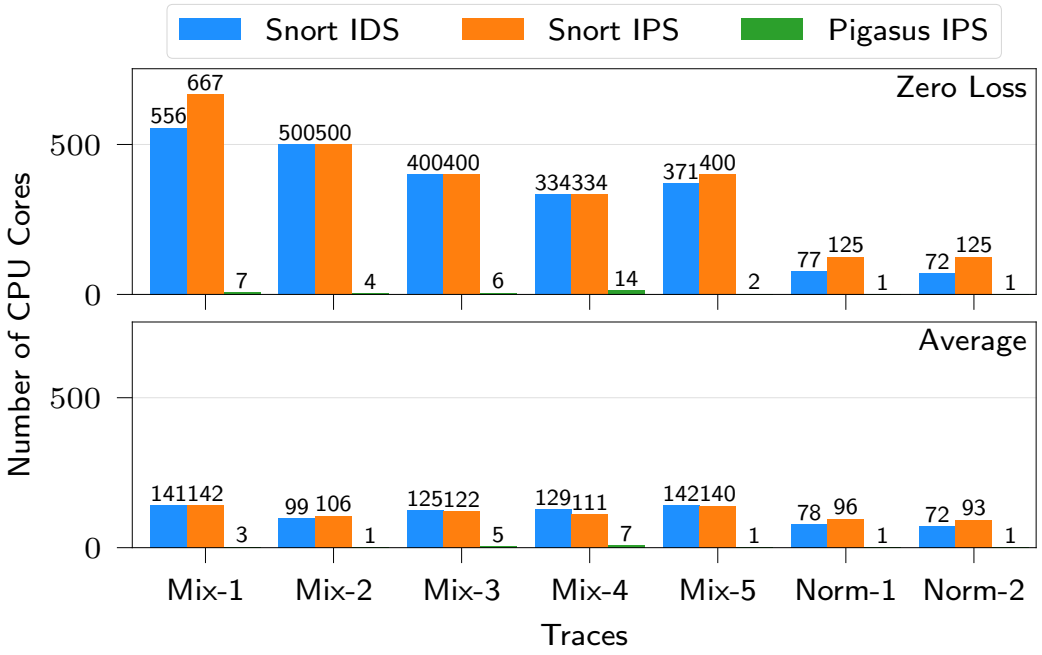


Figure 3.2: Number of cores required to process each trace at 100Gbps using Pigasus (FPGA + CPUs) and Snort (traditional CPUs alone). Pigasus numbers are based on implementation; Snort numbers are extrapolated from its single-core throughput and assume perfect linear scaling.

can only achieve 55Gbps zero-loss throughput due to bursty matches in that trace. Since the hardware datapath only occupies half of the FPGA, we assume that by scaling up the performance bottleneck component (*i.e.*, making that module 2× larger), FPGA will not be the performance bottleneck. For the Snort experiments we run Snort in both IDS and IPS mode (with DPDK) on a single core and increase the throughput until it begins to drop packets. Note that while we report the actual number of cores required to run Pigasus, for Snort we extrapolate the single-core experiment to determine the number of cores that we would need to keep up with 100Gbps. This considers that Snort's throughput scales linearly with the number of cores and, therefore, represents an ideal *lower bound* to the actual number of cores

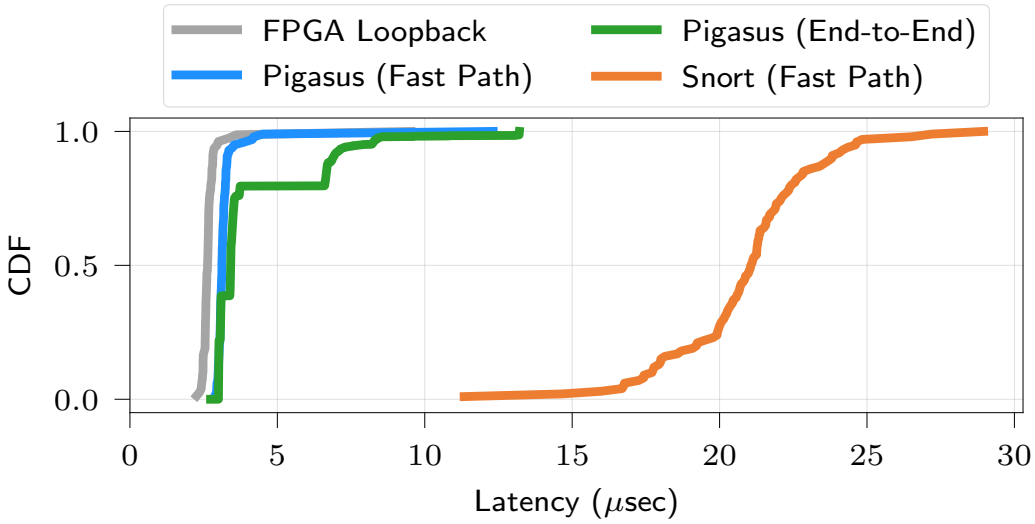


Figure 3.3: CDF of latency of Pigasus vs. Snort

needed to run Snort. Overall, we see that Snort in IDS mode requires $23-185\times$ more cores than Pigasus ($65\times$ on average), and in IPS mode requires $23-200\times$ more cores ($72\times$ on average).

Latency: Of course, in a practical IPS we care not only about throughput/provisioning but also per-packet latency. We plot the distribution of per-packet latency in Figure 3.3. We find that Pigasus yields improvement of almost an order of magnitude in the median latency, and up to $3\times$ improvement in the tail latency. As a point of comparison, we also show the baseline performance of a simple FPGA loopback measurement (*i.e.*, without any processing) and the Pigasus fast-path for packets that do not need further CPU processing. We find that the Pigasus fast-path is very efficient and almost comparable to the baseline. We also find that Pigasus end-to-end latency only deviates substantially from the fast-path for the tail. While we hypothesized some improvements in latency, we were puzzled by the magnitude of the improvement. Investigating why Snort was so much slower revealed that

on average, while Pigasus reduced the latency for the Reassembly (by 6μs), Parser (by 4μs), and the MSPM (by 3μs) as expected relative to software, the additional reduction came from avoiding packet I/O overhead in software (around 5μs).

Power footprint: Figure 3.4 depicts the estimated power consumption required to achieve 100Gbps throughput for three configurations: Snort in IDS mode, Snort in IPS mode, and Pigasus in IPS mode. On the CPU side, we use Intel's Running Average Power Limit (RAPL) interface [48] to measure per-core power consumption in steady-state. To verify its accuracy, we also measured the power utilization using an electricity usage monitor [49] and found consistent results. On the FPGA side, we use the Board Test System [41] (part of Intel's FPGA Development Kit) to measure power dissipation in the FPGA core and I/O shell. We observe that, across all traces, Snort (in *either* mode) has a $13-59\times$ higher power consumption than Pigasus ($34\times$ on average). We further note that the reported wattages for Pigasus represent a conservative estimate; while the total power consumption on the FPGA side is 40W, the core fabric accounts for just 13W, and the remainder is used for I/O (including Ethernet). Conversely, we only charge Snort for power consumed during compute tasks, ignoring other overages (such as Network I/O).

Cost: To estimate the Total Cost of Ownership (TCO), we consider both the capital investment and the power cost for each configuration. To estimate the capital investment, we use the per-core pricing data for the AMD EPYC 7452 CPU (32-core CPU with $68.75 per core)[4]. For Pigasus, we also incorporate the market price of an Intel Stratix 10 MX FPGA [41] ($10K). Assuming that the number of cores needed in practice is between the *Zero Loss* and *Average* in Figure 3.2, we estimate that the capital cost of the CPU-only solution is between $7,922 and $25,045, while

[4]We found the per-core cost of Intel Xeon or i9 processor is within the similar range.

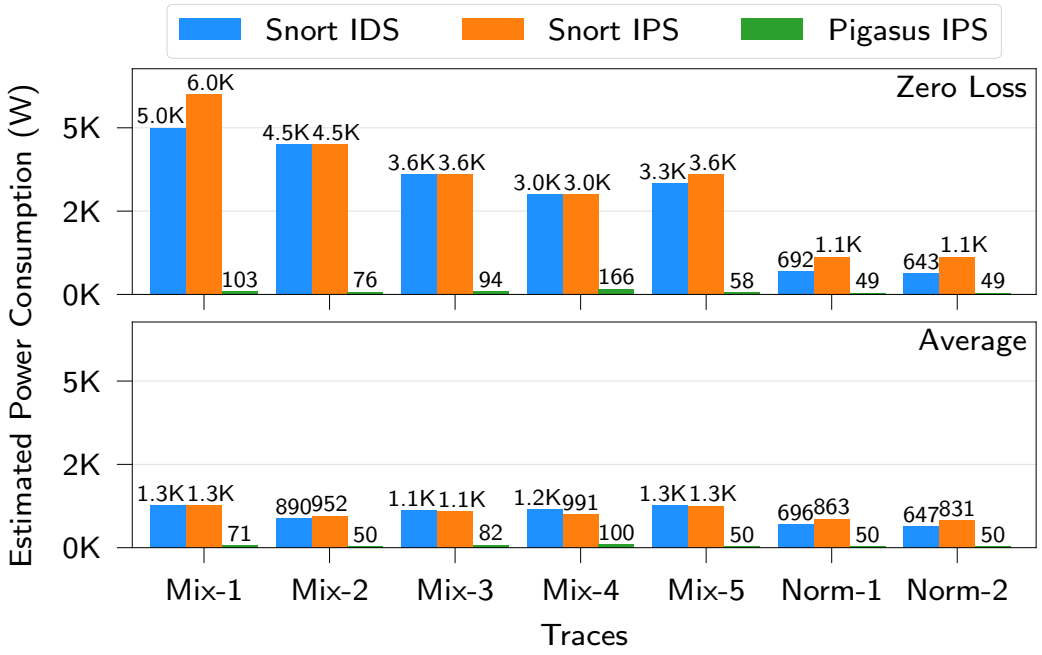


Figure 3.4: Estimated wattage to achieve 100Gbps

the capital cost of Pigasus is between $10,189 and $10,344. To estimate the power costs, we assume a lifetime of 3 years and electricity cost at $0.1334/kWh (average electricity rate in the US [50]). The power cost of the CPU-only solution at 9W/core is between $3,636 and $11,494, while the cost for Pigasus is between $227 and $298. Then, combining the capital investment and the power cost, the TCO of the CPU-only solution is between $11,558 and $36,539, while the TCO of Pigasus is between $10,416 and $10,642, saving between $1,142 and $25,897. We note that these estimates consider retail prices and do not account for other operational costs, such as cooling and rack space, which we expect to favor Pigasus. Moreover, for 100K flows and 10K rules, we only use about half of an Intel Stratix 10 MX FPGA; one may consider adapting the design to a smaller FPGA, further reducing the cost of Pigasus.

Resource breakdown: Table 3.1 shows the FPGA resources used by each component of Pigasus when configuring it to support 100K flows and 10K rules. As we can see, the end-to-end design takes less than half of our board. In particular, the String Matcher and Flow Reassembler use 6MB BRAM, an order of magnitude smaller than existing FPGA designs.

We note that after DSP optimizations[5], Pigasus does not use any DSPs while maintaining almost the same amount of ALM usage compared with our OSDI paper [51]. In our initial implementation of MSPM, we replicated the Hyperscan's multiplication-based hashing algorithm using DSP blocks. While the DSP blocks were not the resource bottleneck and the other Pigasus components were not using them, this implementation was non-ideal, as it missed some optimization opportunities: (1) utilizing the 'shifting window' nature of MSPM to remove unnecessary multiplications by breaking 64B multiplication to per-byte multiplication; (2) implementing the sparse and narrow multiplication using LUTs instead of DSPs. As a result, all of the DSPs blocks were gone; the LUTs that were used to distribute the data among a large number of DSPs were saved (about the same number as those consumed by the new design). The saved DSP resources can be used by other applications in a multi-tenant scenario in the future. The optimization also reduces the routing pressure and thus improves the compilation time by almost 2×.

Recall that our original goal was to achieve 100Gbps supporting hundreds of thousands of flows matching tens of thousands of rules on a single server with a reasonable cost/resource footprint. The above results suggest that Pigasus indeed achieves this goal (with ample headroom).

[5]Optimized by outside contributor Dr. Moein Khazraee.

Module	ALM	BRAM (MB)	eSRAM (MB)
Packet Buffer	507 (0.1%)	0 (0%)	5.91 (50.0%)
Flow Reassembler	20,847 (3.0%)	2.61 (15.6%)	0 (0%)
String Matcher	141,946 (20.3%)	3.36 (20.1%)	0 (0%)
DMA Engine	1,718 (0.3%)	0.32 (1.9%)	0 (0%)
Instrumentation	1,155 (0.2%)	0 (0%)	0 (0%)
Vendor IPs	39,899 (5.7%)	1.14 (6.8%)	0 (0%)
Miscellaneous	10,827 (1.54%)	0.76 (4.6%)	0 (0%)
Full Design	216,899 (30.9%)	8.20 (49.0%)	5.91 (50.0%)

Table 3.1: Resource breakdown. Percentages are relative to the total amount of resources in a Stratix 10 MX FPGA.

3.3.3 Microbenchmarks and Sensitivity Analysis

In this section, we present Pigasus' performance sensitivity to traffic characteristics. We probe deeper into Pigasus' performance under differing levels of malicious traffic. We further characterize the performance impact of packet size.

Dependence on CPU offload: We construct semi-synthetic traffic traces by mixing malicious flows extracted from *mix-1* trace with innocent trace *norm-2* in different proportions.[6] Figure 3.5(a) depicts the dependence of zero-loss throughput on the fraction of malicious flows (in terms of relative packet count). We report results for Pigasus (using both 1 and 16 cores) and Snort IPS (with 1 core). We observe that, as long as the percentage of malicious traffic is less than 15%, Pigasus is able to process packets at line-rate using a single CPU core. With 16 cores, Pigasus can process packets at line-rate for up to 50% of malicious traffic. After the 50% mark, performance begins to degrade gradually. We repeated the same experiments disabling the software component of Pigasus and observed that the throughput matches the 16-core experiment, suggesting that the hardware is the bottleneck. More specifically, the MSPM's rule reduction logic is stressed by the

[6]Note that not every packet in a malicious flow triggers a match.

large number of potential rule matches.

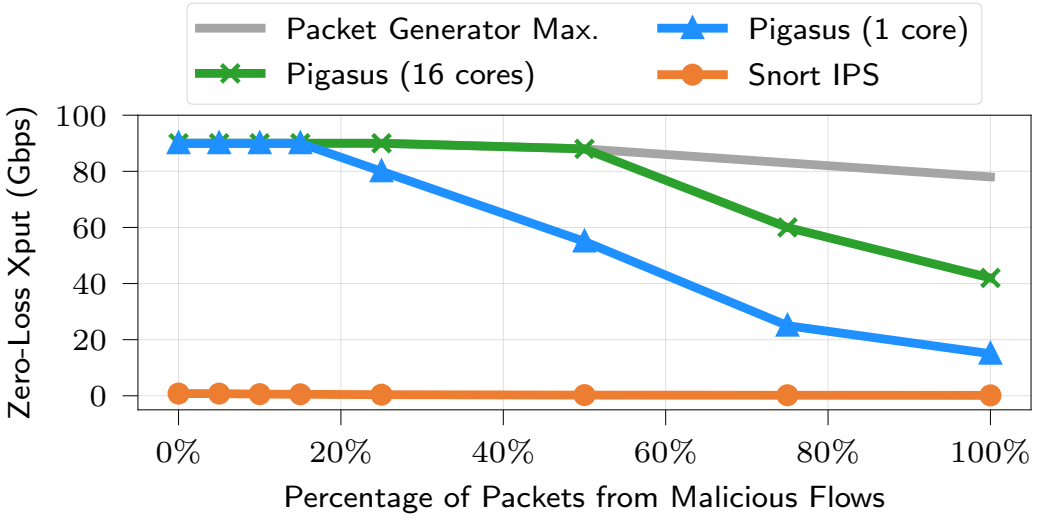
Figure 3.5(b) depicts the number of cores required to achieve 100Gbps as a function of the fraction of packets from malicious flows for up to 50%. Results for Snort are extrapolated from the single-core throughput. Despite the performance degradation observed in (a), Pigasus scales considerably better than Snort, requiring two orders of magnitude fewer cores. We also note that, while the hardware only becomes the bottleneck at an extreme fraction of malicious traffic, the design can be made even more robust using two hardware pipelines (discussed further in Section 3.3.4).

Dependence on packet size: We consider the impact of packet size on Pigasus' performance stemming from the linked-list-based TCP Reassembler design. We configure the Moongen packet-generator to generate fixed-sized synthetic packets, and measure end-to-end, zero-loss throughput as we vary the packet size. Figure 3.6 illustrates this dependence in terms of both Gbps and millions of packets per second. We observe that, for packets exceeding 500B (comparable to average packet sizes on the Internet [52]), Pigasus is capable of processing at line-rate. (More generally, Pigasus, by design, can sustain 100Gbps as long as the average packet size is greater than 500B over a window of 87μs estimated based on buffer size.)

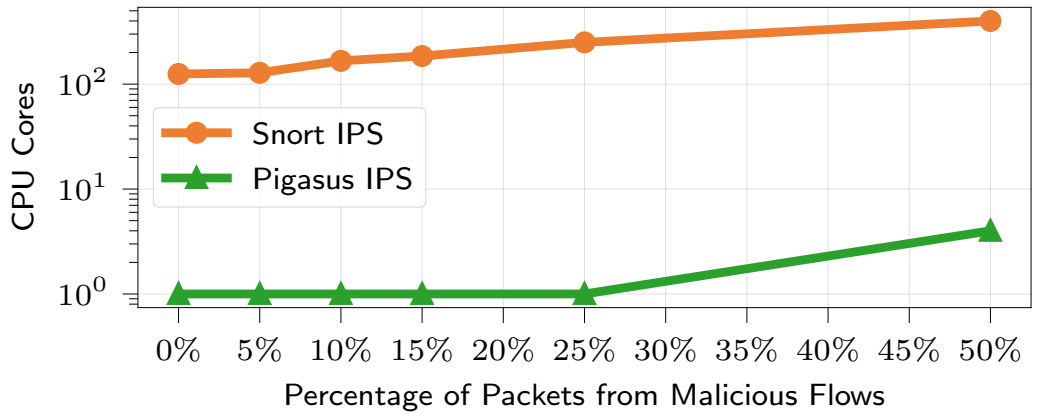
3.3.4 Future Outlook

Supporting 100Gbps with 100K flows and 10K rules requires only about half of the resources in our FPGA. We now explore what we can do with the additional capacity.

One option is to duplicate the existing processing pipeline (which runs at 100Gbps/25Mpps) each to serve a different subset of flows, increasing the through-

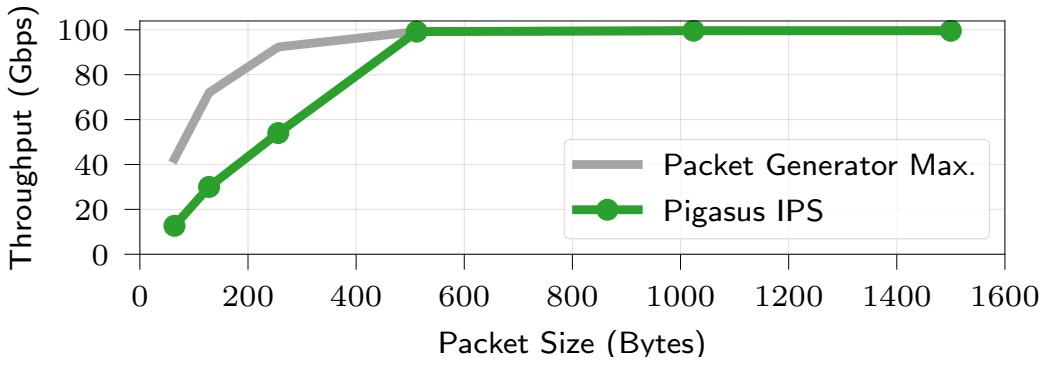


(a) Zero-loss throughput.

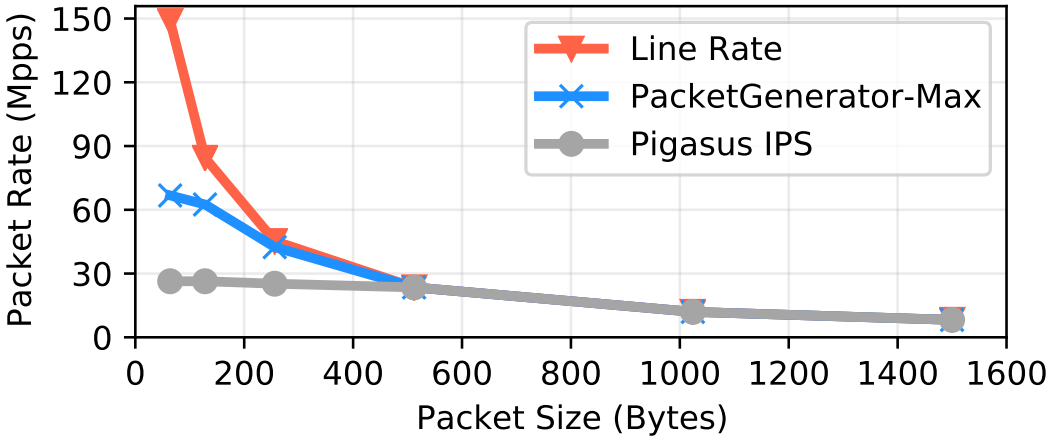


(b) Number of cores required to achieve 100Gbps.

Figure 3.5: Impact of the fraction of malicious traffic on system throughput



(a) Zero-loss throughput.



(b) Packet rate.

Figure 3.6: Zero-loss throughput and Packet rate (in millions of packets per second) achieved by Pigasus for a range of packet sizes

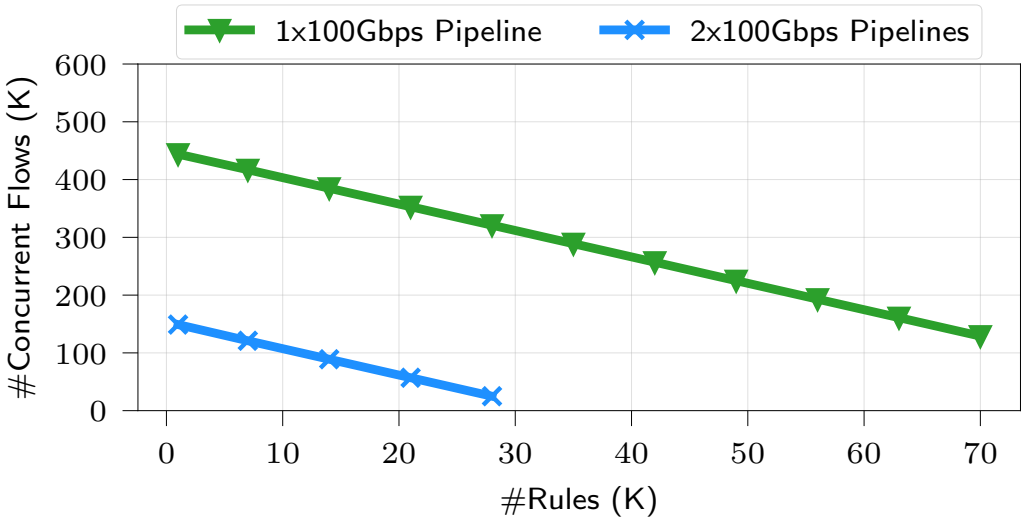


Figure 3.7: Tradeoff between the number of supported rules and concurrent flows when using one or two 100Gbps hardware pipelines

put to 200Gbps, at the cost of creating additional copies of all the MSPM engines. Another option is to increase the number of supported flows or rules. Figure 3.7 depicts the three-way tradeoff between the scalability of the number of rules, concurrent flows, and replicated hardware pipelines. The design with two pipelines benefits from better throughput but has less room for storing rules or flows. There is plenty of scaling headroom in the Pigasus FPGA frontend design for more rules and flows. In Chapter 7, we will show how we easily explore the design flexibility.

Chapter 4

FPGA-First Architecture

After introducing the context of our Pigasus artifact in Chapter 3, we now present the key insights behind Pigasus. In particular, Chapters 4 (the current chapter), 5, and 6 together demonstrate our first insight – tune for the common case, while handling the uncommon case in resource-efficient ways. In this chapter, we present the design decision made at the end-to-end system level – *FPGA-first architecture*, where we tailor the design for common traffic to achieve both high-performance and low resource consumption simultaneously. This chapter is organized as follows: we first describe the FPGA-first architecture and why it should be used; after summarizing the principles of splitting a system between FPGA and CPU, we evaluate the idea and briefly discuss the new challenges this FPGA-first architecture introduces.

4.1 What is FPGA-First Architecture?

Traditionally, FPGA acceleration for streaming applications generates static designs with fixed performance. Since these designs are over-provisioned for the worst-case scenario, they are too resource-consuming to be mapped on a single FPGA in the

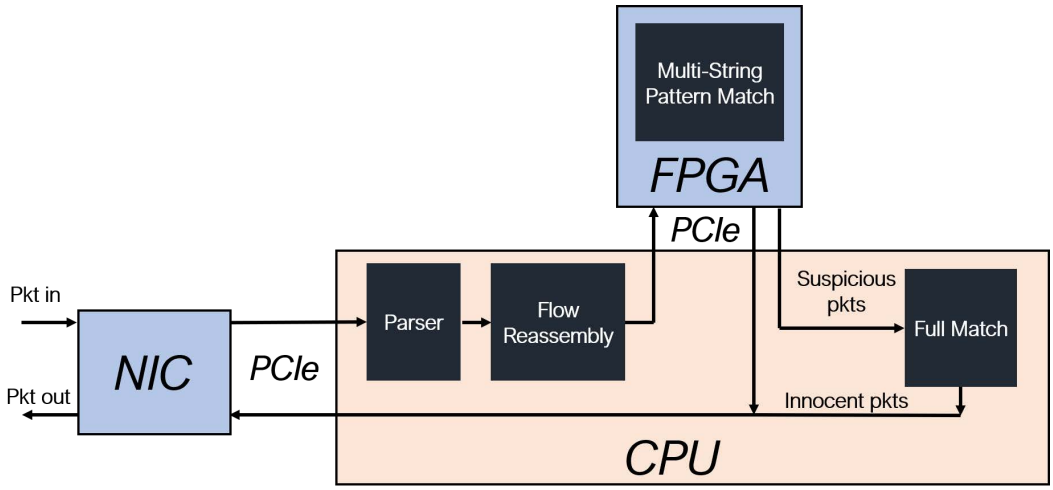


Figure 4.1: Traditional FPGA-as-offload scheme

context of IDS/IPS. Therefore, most of the existing FPGA-accelerated IDS/IPS work [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] follows what we call an *FPGA-as-offload* scheme, where the CPU is still the *primary* processing unit, and the FPGA works as the *offloader* to CPU, accelerating *a single task* in an *input-independent* manner. Figure 4.1 illustrates an example where the MSPM task is accelerated by the FPGA offloader. The traffic goes to the CPU first through the NIC. The CPU performs *parsing*, *TCP reassembly*, and then offloads the MSPM task to the FPGA through PCIe. Once the packets are processed by the FPGA, they are sent back to the CPU for further processing. Unfortunately, as we will show in Section 4.2, accelerating a single task cannot improve the end-to-end system performance by an order of magnitude.

FPGA-first architecture reverses the traditional roles of FPGA and CPU. FPGA is now the *primary* compute platform – performing the majority of the work – and the CPU is *secondary*, operating only as needed. More specifically, the FPGA should have direct access to the input data and process the common cases, leaving

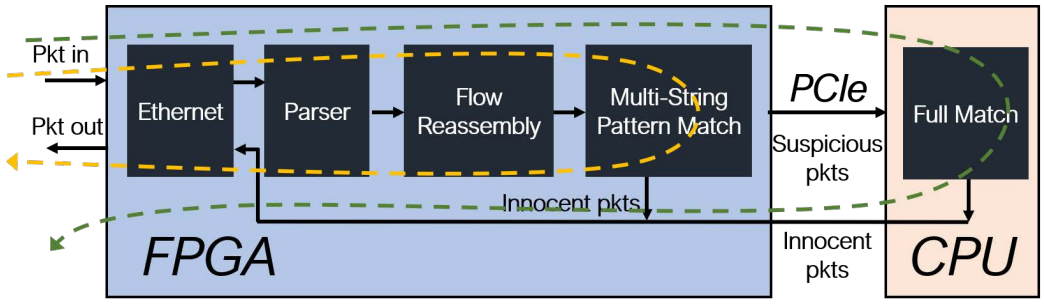


Figure 4.2: FPGA-first architecture (yellow dashed line represents the common path; green dashed line represents the uncommon path)

the uncommon cases to the CPU, which now works as a complexity offloader. Figure 4.2 depicts FPGA-first architecture in the context of IDS/IPS, where *all packets and all rules* (common case) are processed in-line by the highly parallel FPGA datapath while the *suspicious packets with a few partially matched rules* (uncommon case) are sent to the CPU for full evaluation.

A key difference between FPGA-first architecture and other FPGA in-line processing [1, 30] is that in FPGA-first architecture, FPGA itself is also the *control* unit. In most existing FPGA in-line processing work, the CPU is still the *control* unit that manages the flow states, steers traffic, and configures the 'dumb' FPGA datapath. Since complex stateful operations like TCP reassembly have to be done on the CPU, this model fundamentally limits the use of FPGAs – it sits in-line but only works on simple and input-independent tasks such as checking IP addresses or manipulating headers. In contrast, in the FPGA-first architecture, the FPGA itself is responsible for managing the flow states and dynamically steering traffic among streaming kernels, while the CPU is only responsible for handling the rarely triggered complex corner cases in a *stateless* fashion.

4.2 Why FPGA-First Architecture?

The FPGA-first architecture can balance the two conflicting requirements – *high performance* and *low resource utilization* – by accelerating the common case entirely on FPGA and offloading the uncommon case to CPU. Such balance could not be achieved by either fixed-performance worst-case-oriented design or single-task offloading on FPGA.

A straightforward way of handling input-dependent behaviors is to provision for the worst case, such that a single design can meet the requirements under all circumstances. For instance, one may want to map all of the tasks to FPGA, disposing the need for CPUs. Nevertheless, this worst-case-oriented design is infeasible, as it provides few performance benefits compared with the common-case-tailored design at a *high resource cost* in terms of memory.

For example, as we will see in Chapter 6, the Full Match stage only interacts with ≈5% of packets in Pigasus. Hence, Full Match stage is not a performance bottleneck for the majority of packets. Furthermore, regular expression parsing is a very mature research area, and yet state-of-the-art hardware algorithms do not reach our performance and memory demands for Pigasus. We estimate that Grapefruit [42], a state-of-the-art regular expression engine for FPGAs, would require 8MB of BRAM to statically map all the regular expressions from our ruleset on the FPGA, and yet would still only keep up with a few Gbps of traffic. Hence, we would likely need multiple replicas of the Grapefruit design – at least 24MB of BRAM – to keep up with the average of 5Gbps of traffic that reach the Full Matcher. Therefore, offloading regular expressions would exhaust our memory budget for little gain, in that the majority of packets will never execute the Full Matcher anyway.

On the other hand, as the worst-case-oriented designs are too expensive to

be practical, most FPGA-accelerated IDS/IPS work focuses on creating accelerators for a single task [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. Unfortunately, a basic analysis based on Amdahl's law reveals why this approach fundamentally fails in achieving *high performance*.

Figure 4.3 illustrates the fraction of CPU time spent on each task in Snort 3.0: Parsing, Reassembly, Multi-String Pattern Match, Full Match, and other tasks. As we can see, no single task dominates CPU time – at most, MSPM consumes 46% of CPU time for the Mixed-4 trace. Note that MSPM used to be the performance bottleneck before the appearance of Hyperscan pattern-matching library. Different from the original state-machine-based Aho-Corasick algorithm [31], Hyperscan adopts a SIMD-accelerated ShiftOR filter + Hash Table filter solution, which significantly improves the MSPM performance, yielding 8× end-to-end speedup on modern CPUs [2]. Because of the significant improvement of Hyperscan, there is little opportunity to improve the end-to-end speed by just accelerating a single task. Using Amdahl's law, we can see that even if MSPM were offloaded to an imaginary, infinitely-fast accelerator, throughput would increase by only 85%, far from our two orders of magnitude target.

The key lessons are (a) we should identify and accelerate the common case at *end-to-end system* level, and (b) we should avoid worst-case-oriented design when the performance gain cannot justify the cost of over-provisioning resources.

4.3 Principles of FPGA/CPU Splitting

Now, the natural question is which tasks should remain on the CPU? The general principles are (a) the task should be less performance-critical, *i.e.*, it takes a short time to run on CPU and/or is triggered infrequently, (b) the task should be at the

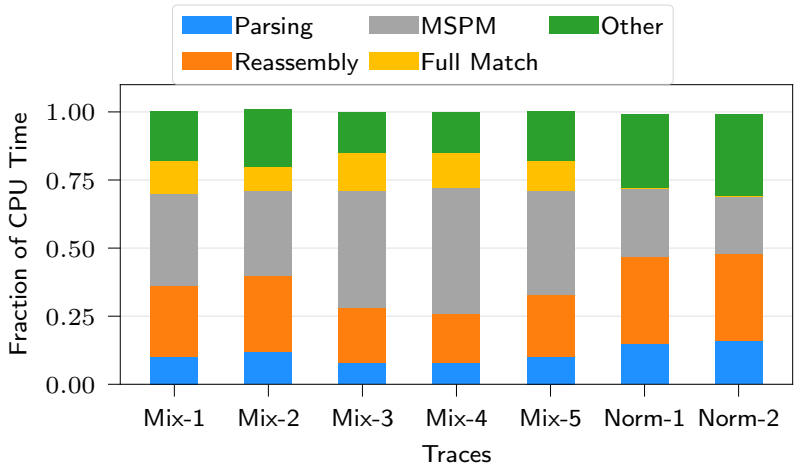


Figure 4.3: Fraction of CPU time spent performing each task in Snort 3.0 with Hyperscan.

end of the processing pipeline to avoid multiple FPGA-CPU data movements, and (c) the task should be expensive to implement on FPGA.

Following the above principles, we identify *Full Match* to be a good candidate task. Full Match is only triggered by ≈5% of packets with ≈2.6 rules/packet, which is clearly the uncommon case. As each packet-rule pair is processed sequentially, it also fits the CPU's von Neumann architecture. Furthermore, Full Match is the last stage, so there will be no multiple round-trips between FPGA and CPU. Most importantly, as we already discussed in Section 4.2, it is expensive to implement it on the FPGA. That is why we decided to split the end-to-end system at the Full Match stage.

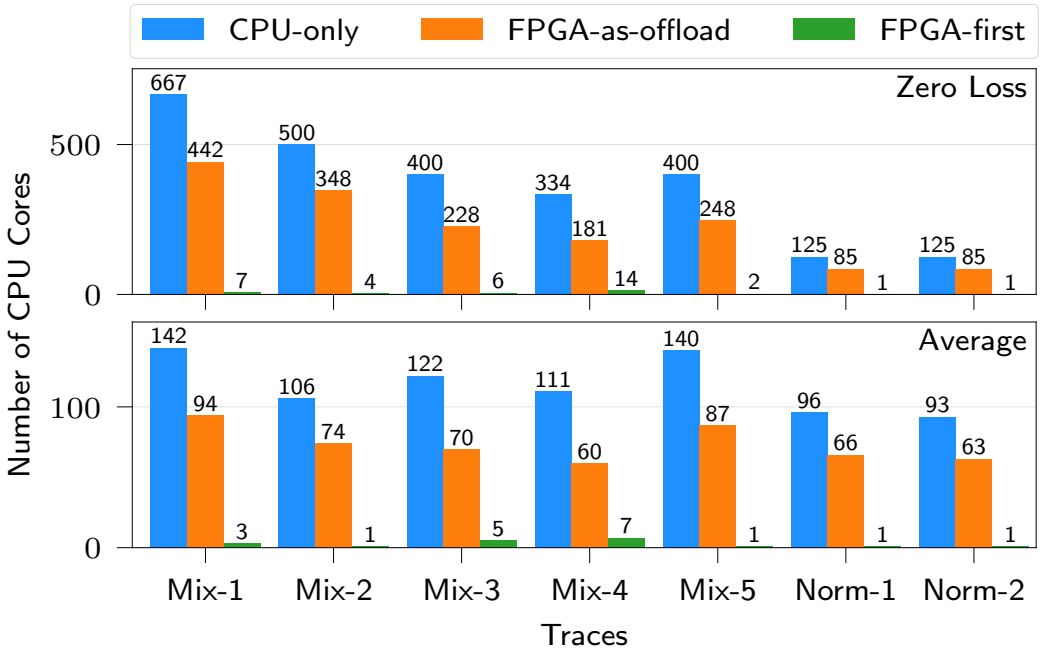


Figure 4.4: Number of cores required to process each trace at 100Gbps for different designs.

4.4 Evaluation

Setup: To evaluate the effectiveness of FPGA-first architecture, we conduct two experiments. In the first experiment, we compare the performance of the single-task offload – 'FPGA-as-offload' – with our FPGA-first architecture. In the second experiment, we compare the cost of the worst-case-oriented design – 'FPGA-only' – with the FPGA-first architecture. We use the same setup as described in Section 3.3.1. We also plot the 'CPU-only' (Snort baseline) as a reference, which was discussed in Section 3.3.2.

Comparing performance of FPGA-as-offload and FPGA-first: Figure 4.4 reports the number of cores needed to reach 100Gbps for different traces. The *CPU-only* bar refers to Snort IPS, and the *FPGA-first* bar refers to Pigasus IPS.

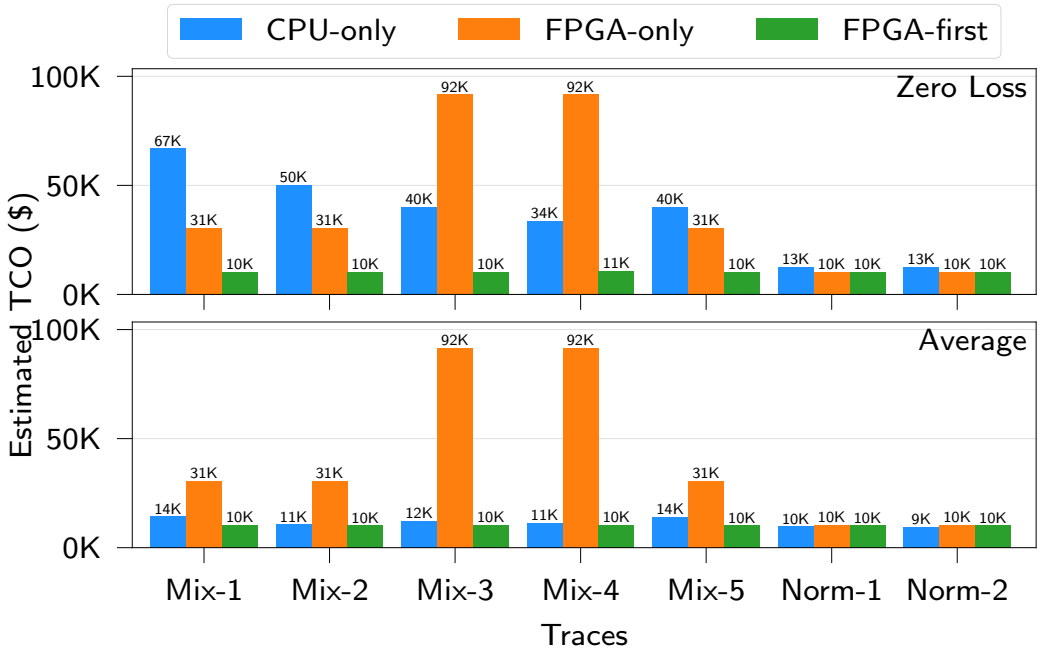


Figure 4.5: Estimated total cost of ownership (TCO) of different designs to reach 100Gbps

The *FPGA-as-offload* bar shows the extrapolated number of cores assuming max single-task speedup. Specifically, we assume the performance bottleneck for each trace shown in Figure 4.3 can be accelerated by an infinitely-fast implementation without any FPGA-CPU communication overhead. As we can see in Figure 4.4, even with this generous assumption, the *FPGA-as-offload* approach still has a huge gap to reach 100Gbps with a single server, using $9 - 124\times$ more cores than our *FPGA-first* approach.

Comparing cost of FPGA-only and FPGA-first: The worst-case-oriented design ('FPGA-only') assumes everything can be implemented on FPGAs, which can theoretically match the performance of *FPGA-first* without using any CPU cores. Figure 4.5 plots the estimated total cost of ownership (TCO) of *CPU-only*,

FPGA-only, and *FPGA-first*. The TCO calculation follows the same steps as in Section 3.3.2. The *FPGA-only* is more complicated. Despite the design complexity, we assume the entire Full Match logic can be implemented on FPGA with 8MB BRAM running at 1.6Gbps based on our estimation of Grapefruit design. For each of the traces, depending on the different performance requirements of Full Match, we may need different numbers of FPGA Full Match instances. If the total resources exceed the board capacity, we assume we can allocate multiple FPGAs.

From Figure 4.5, we can see that as expected, *FPGA-first* costs much less than *FPGA-only*, on average saving $31,900. The *FPGA-only* approach provisions for the worst case, so the cost does not change between *Zero Loss* and *Average*. Some traces (*i.e.*, Mix-3, Mix-4) invoke the Full Matcher more frequently (up to 15.9% of traffic), and hence require more FPGA Full Match instances to keep up, resulting in significant cost increase. For other traces like Norm-1 and Norm-2, where less than 1% of traffic reach the Full Match stage, the FPGA Full Match stage can coexist with the rest of the Pigasus pipeline on one FPGA, which explains the same TCO of the *FPGA-only* and *FPGA-first*. However, given that the rest of the Pigasus pipeline only occupies half of the board in *FPGA-first*, one may want to map Pigasus to a middle-range FPGA, with half of the $10,000 price, or double the performance with the extra space. Either way would be more cost-effective than the *FPGA-only* approach.

Discussion: In summary, compared with *FPGA-as-offload*, the *FPGA-first* architecture makes 100× speedup possible by accelerating the majority of the processing. Compared with *FPGA-only*, the *FPGA-first* architecture chooses to use a few CPUs to handle uncommon cases, saving expensive and scarce FPGA resources. However, this system-level design decision introduces new challenges. We now need to imple-

ment most of the tasks on FPGAs and run them at 100Gbps. This is *impossible* if we reuse existing FPGA-based TCP reassembly and MSPM designs due to high BRAM consumption. In the next two chapters, Chapter 5 and Chapter 6, we will show how we achieve the target performance with minimal resources for TCP reassembly and MSPM, respectively.

Chapter 5

Fast-Slow Path SRAM-Based TCP Reassembly

A natural consequence of FPGA-first architecture (described in Chapter 4) is that we now need to support TCP Reassembly on FPGA. In this chapter, we first describe the requirements of TCP Reassembly and the design space. We then present our fast-slow path SRAM-based design before evaluating it.

5.1 TCP Reassembly Requirements

Reassembly refers to the process of reconstructing a TCP bytestream in the presence of packet fragmentation, loss, and out-of-order delivery. Reassembly is necessary for IDS/IPS because the MSPM and Full Matcher must detect patterns (strings or regular expressions) that may span across more than one packet. Figure 5.1 depicts an illustrative example where the string pattern 'Attack' spans across the TCP packet boundary. TCP Reassembly can reassemble Packet 1 and Packet 2 – even when they arrive out of order – such that the MSPM and Full Matcher can still

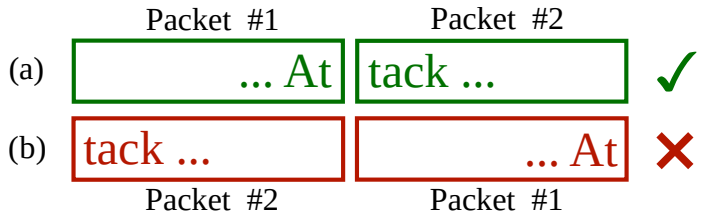


Figure 5.1: Illustrative example of TCP Reassembly

detect the string pattern 'Attack.' However, without TCP Reassembly, the out-of-order (OOO) packets where Packet 2 (with 'tack') comes before Packet 1 (with 'At') will successfully bypass IDS/IPS without getting detected. As a consequence, they will get reassembled at the end-host machine and the attack will be triggered. This was a typical evading technique before TCP Reassembly becomes an essential component of IDS/IPS [53].

Note that our goal is not to implement a full TCP endpoint, and hence we are not responsible, *e.g.*, for producing ACKs; the IDS/IPS is a passive observer of traffic between two existing endpoints, merely reordering the packets it observes for analysis. The key objective of our Reassembler is to perform this reordering for hundreds of thousands of flows while operating at 100Gbps, *within the memory limitations* of our FPGA.

5.2 Design Space for TCP Reassembly

Hardware design often favors data structures that are *fixed-length*, *constant-time*, and generally *deterministic*, and most TCP Reassembly designs follow suit. For instance, Yuan *et al.* [29] allocates a fixed 64KB packet buffer in DRAM and uses 7 pairs of pointers to track OOO state for each flow; similarly, Sidler *et al.* [28] maps a fixed-sized 'segment array' in DRAM to track per-flow state. By using

static buffers, these designs can achieve nearly constant-time insertion of out-of-order packets into memory; furthermore, the memory consumed by any individual flow is fixed, so freeing space is also deterministic. In addition, each flow is bounded in its resource consumption, and so a highly out-of-order flow cannot take over the available address space, starving other flows.

The above designs have been working well for 1Gbps or 10Gbps line-rate network systems. However, as the line-rate reaches 100Gbps and keeps increasing rapidly, DRAM-based packet buffering is struggling to keep up. A 100Gbps network system needs to guarantee simultaneous 100Gbps write and 100Gbps read constantly. That is 200Gbps, which is exactly the theoretical peak bandwidth of a modern DDR4 3200 component (our board has a slower DRAM component, DDR4 2666). Considering that the effective bandwidth of DRAM is always lower than the peak bandwidth, we need a different approach to buffer the packets.

One potential approach is to keep the same data structures but develop a logical single-address-space memory with guaranteed bi-directional 200Gbps bandwidth. This is possible if we aggregate multiple channels (each channel has its own address spaces) of DRAMs or even High Bandwidth Memory (HBM) for enough bandwidth and use SRAM buffers in front to deliver rigid 200Gbps read/write bandwidth to address the non-determinism of DRAM technology. However, multi-channel DRAM and HBM may not be available to many FPGA boards, and it is very complicated to balance between read/write accesses, different channels, and SRAM/DRAM while delivering guaranteed performance.

We instead take the SRAM approach to leverage its high and deterministic bandwidth. Unfortunately, simply adopting the statically allocated fixed-size buffer approach does not work for SRAM as it both *wastes memory* and *limits out-of-order*

flows. For example, allocating 64KB for *each and every flow* [28] would require 64MB to support 1K OOO flows (typically 1% of the total 100K flows [54]) – an order of magnitude bigger than our FPGA on-chip memory capacity. On the other hand, flows that do suffer a burst of out-of-order packets (perhaps due to network loss) that exceeds the 64KB capacity cannot be served, even if there is memory available. Therefore, we need a more efficient way to utilize the limited on-chip memory resources.

For software developers, the obvious response to these challenges is to use a more memory-dense data structure such as a linked list, where each arriving segment is allocated on-demand and inserted into the list in order. Because memory is allocated on demand, no memory is wasted, and those flows which need more capacity can consume more as available. In our empirical traces, 0.3% of packets arrive out-of-order, with 'holes' in the TCP bytestream typically filled in after 3 packet arrivals from the same flow. In a linked-list-based design, this means that on average an out-of-order flow consumes 5K bytes at most.

From a hardware perspective, however, a linked list is an unorthodox choice: pipeline parallelism depends on each stage of the pipeline taking a fixed amount of time. Since linked lists have variable insertion times, depending on how far into the list a segment must be inserted, linked lists can lead to pipeline stalls which result in non-work-conserving behavior upstream from the slow pipeline stage, and hence overall poor throughput. We find that by carefully designing the reassembly pipeline as a combination of a *fast path* (only handling the common-case in-order packets) and a *slow path* (that handles the uncommon-case out-of-order packets), one can achieve the best of both worlds.

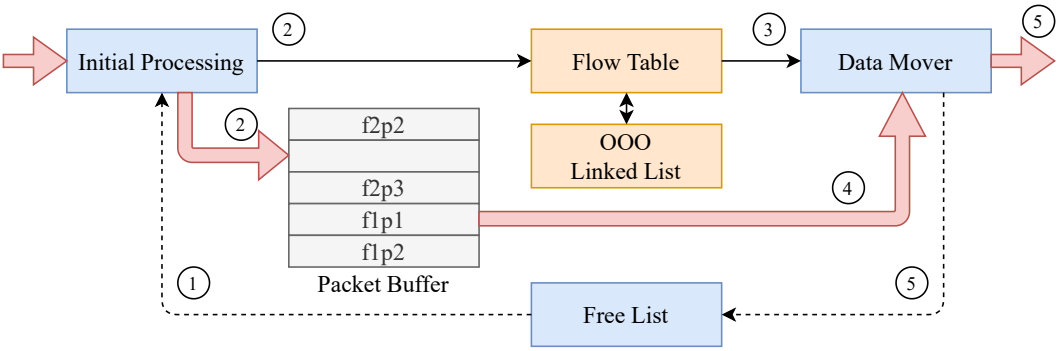


Figure 5.2: Reassembly overview

5.3 Fast-Slow Path SRAM-Based Design

In this section, we describe our fast-slow path SRAM-based design. We first give an overview of the design and show how packets trigger different operations. We then describe in detail our three execution engines and flow table implementation.

5.3.1 Overview

Figure 5.2 presents the overview of our Reassembly design. When a packet enters Reassembly, it will first fetch an unused 'packet ID' from the *Free List*, which tracks the free slots in the *Packet Buffer*. Then, the *Initial Processing* unit will store the packet to the 2KB (bigger than the 1500B Maximum Transmission Unit of a packet) fixed slot indexed by the 'packet ID' in the Packet Buffer implemented in eSRAM[1]. Note that the Packet Buffer is shared by all flows. At the same time, the extracted metadata[2] will be sent to the *Flow Table*. The *Flow Table* and *OOO Linked List* (described more thoroughly in Section 5.3.3 and Section 5.3.2) will send the sorted packet metadata to the *Data Mover*, which then fetches the actual packet from the

[1]We also provide a BRAM-variant design in case the target FPGA board does not have eSRAM.
[2]Metadata extracts packet information for easier processing, including 5-tuple, sequence number, packet ID, *etc.* Each metadata is about 200 bits that can be moved around in a single cycle.

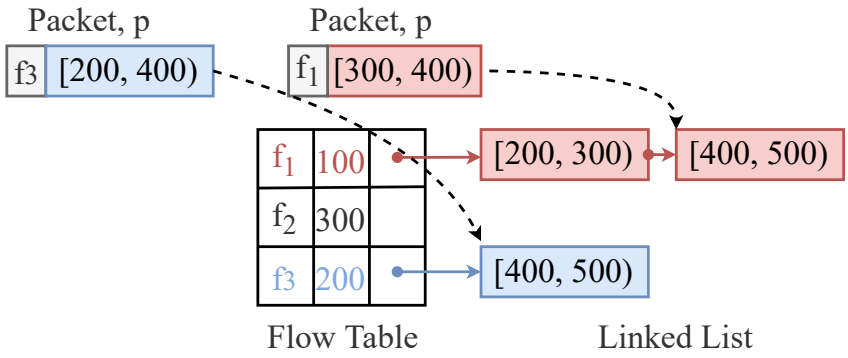


Figure 5.3: Simplified examples of reassembly functionality

Packet Buffer using 'packet ID' and releases the 'packet ID' to the *Free List* for recycling while forwarding the packet to the MSPM stage. This decouples *per-flit*[3] data movement from the *per-packet* (or per-metadata) processing, giving more cycle budget to the core reassembly logic, *i.e. Flow Table*, *OOO Linked List*.

Figure 5.3 illustrates how we handle scenarios of different packet orders. The *Flow Table* is a Hash Table implemented in BRAM, mapping the classic flow 5-tuple identifier to a table containing (a) the *next expected sequence number* for an in-order packet, and (b) the *pointer* to the header node for a linked-list. Note that the *OOO Linked List* is implemented in a separate BRAM, and its space is shared dynamically by all of the flows. A linked-list node only stores the metadata (including 'packet ID') of an *out-of-order* packet.

For example, in Figure 5.3, *flow 1* is an out-of-order flow that has the pointer to the linked-listed nodes and is waiting for the packet with sequence number '100.' However, the incoming packet's sequence range is 300-400, which will invoke linked-list traversal before getting inserted. An in-order flow, such as *flow 2*, does not

[3]Flit is defined as the data unit processed by the FPGA datapath *at one cycle*. Flit size depends on the design. In Pigasus, one flit is 512-bit or 64B. A packet may have multiple flits and hence may take multiple cycles to move around.

consume any storage capacity of the *OOO Linked List* and *Packet Buffer*. If the first 'hole' of an out-of-order flow is filled (*e.g. flow 3*), then the linked-list node will be released and the *next expected sequence number* will proceed to the last released packet(*e.g.*, 500 for *flow 3*). The released space of the *OOO Linked List* can be dynamically used by new OOO packets by using the free list again. If the released node is the last in the linked list, then this flow becomes in order.

5.3.2 Three Execution Engines

To avoid pipeline stalls due to variable-time packet insertions, Pigasus uses *three execution engines* to manage the reassembly state, each of which handles a different class of incoming packet metadata. The *Fast Path* processes in-order packets for established flows; the *Insertion Engine* handles SYN packets for new flows; and the *OOO Engine* handles OOO packets for existing flows. Because Pigasus is implemented in hardware, these engines can all run simultaneously (on different packet metadata) without stalling each other, but must be careful not to conflict in accessing the shared state in the Reassembly *Flow Table*. The flowchart in Figure 5.4 describes the sequence of steps that occur when packet metadata arrives at the Reassembler.

Fast path: Upon arrival from the parser, each packet header is picked up by the Fast Path, which looks up the flow's entry in the *Flow Table*. If no entry exists for that flow, the Fast Path pushes the packet metadata onto a queue for the Insertion Engine and moves on to the next packet. If there exists an entry for that flow, but (a) the packet metadata does not match the next expected sequence number in the *Flow Table*, or (b) the *pointer* in the *Flow Table* is not null, the Fast Path pushes the packet metadata on to a queue for the OOO Engine. Finally, if the packet metadata

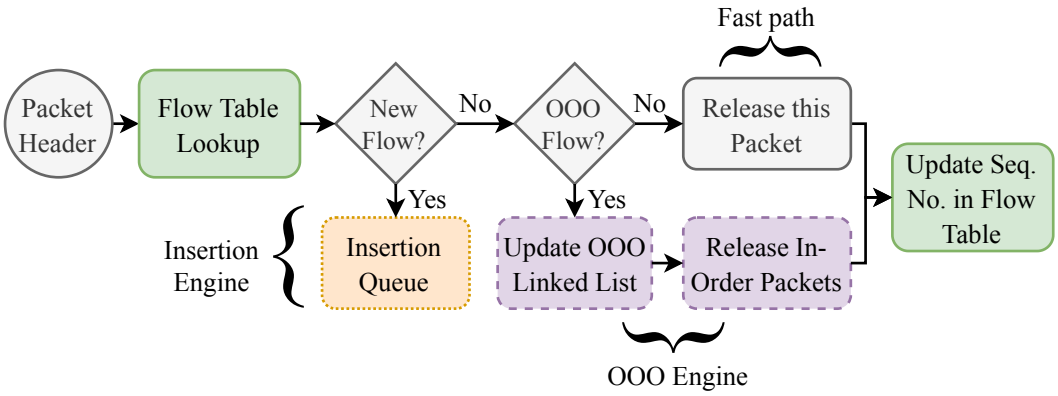


Figure 5.4: Reassembly Pipeline

does match the next expected sequence number in the flow, the Fast Path updates the expected sequence number in the *Flow Table* to the sequence number for the subsequent packet in the flow and pushes the current packet out towards the MSPM. Every task on the Fast Path runs in constant time, and so throughput is guaranteed through this engine to be 25 Million packets-per-second, which amounts to at least 100Gbps, so long as the average packet size is greater than 500B (Internet traces typically have an average packet size of more than 800B [52]).

OOO engine: The OOO Engine does not run in constant time, instead dequeuing packets provided for it from the Fast Path as it finishes operating over the previous packet. For each dequeued packet, the OOO engine allocates a new node representing the packet's starting and ending sequence numbers, traverses the linked list for that flow, and inserts the newly allocated node at the appropriate location. If the packet fills the first sequence number 'hole' in the linked-list, then the OOO Engine removes the now-in-order packet headers from the list, releases them to the MSPM, and also updates the *Flow Table* entry with the new linked-list head and next expected sequence number. If all of the linked-list nodes are released, then this flow becomes in-order, and the *pointer* field of the entry becomes null.

Insertion engine: The Insertion Engine inserts new flow entries into the *Flow Table*; like the OOO Engine, this path too can take variable time. We discuss the Insertion Engine in more detail in the next subsection.

Overall, allocating memory on-demand avoids memory wastage and enables Pigasus to better serve OOO flows that do have a higher memory requirement. Additionally, bifurcating the reassembly pipeline into *fast* and *slow* paths prevents out-of-order flows – which require non-deterministic amounts of time to be served in our design – from impacting the performance of in-order flows, which represent the common case.

5.3.3 Flow Table Implementation

While the Fast Path, Insertion Engine, and OOO Engine all operate simultaneously, they must synchronize over shared flow states (for instance, to keep the next expected sequence number for each flow consistent). We briefly discuss the implementation of our *Flow Table* that provides fast and safe concurrent access to these three engines.

The flow table design borrows a key data structure from FlowBlaze [55]: an FPGA-based Hash Table that employs de-amortized cuckoo hashing [56, 57]. We illustrate this data structure in Figure 5.5. The design provides high memory density (up to 97% occupancy using 4 or more sub-tables [55, 57, 56]), and worst-case constant-time reads, writes, and deletions for existing entries. It also guarantees that, for an Insertion Queue whose size is logarithmic in the number of flow table entries (in practice, a small value), the queue will not overflow [56]. We implement the Hash Table using dual-port BRAM, and the Insertion Queue using a parallel shift-register (capable of storing 8 elements).

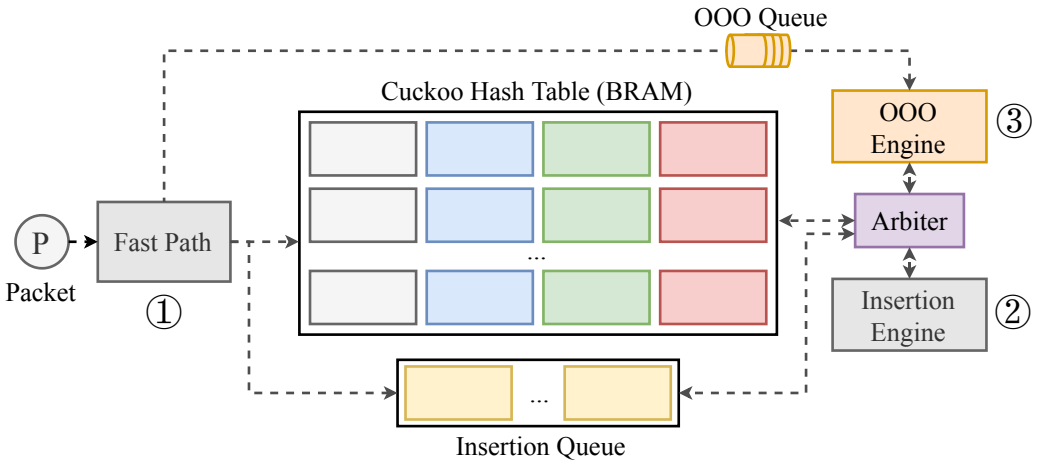


Figure 5.5: Flow Table and OOO Engine

The key to maintaining the Hash Table's de-amortization property is the Insertion Engine, which is responsible for inserting: (a) new flows, and (b) flows that were previously evicted from the Hash Table during a 'cuckoo' step. Effectively, the Insertion Engine dequeues an element from the front of the Insertion Queue and attempts to insert it into the Hash Table. If at least one of the 4 corresponding Hash Table entries is unoccupied, it simply updates the flow table and proceeds to the next queued element; otherwise, it *evicts* one of the 4 flow table entries at random, pushes the evicted entry onto the queue, and inserts the dequeued element in its place.

To guarantee conflict-free flow table access, we have the following prioritization of operations to the table. First, note that the Fast Path and OOO Engine never conflict over the same entry – the flow is either in order, or it is not. The Insertion Engine *can* conflict with both the Fast Path and OOO Engine, as it may try to 'cuckoo' entries. Hence, we enforce the following priorities: (1) Fast Path > Insertion Engine (to ensure deterministic performance on the Fast Path), and (2) Insertion Engine > OOO Engine (to ensure that the queue drains and since,

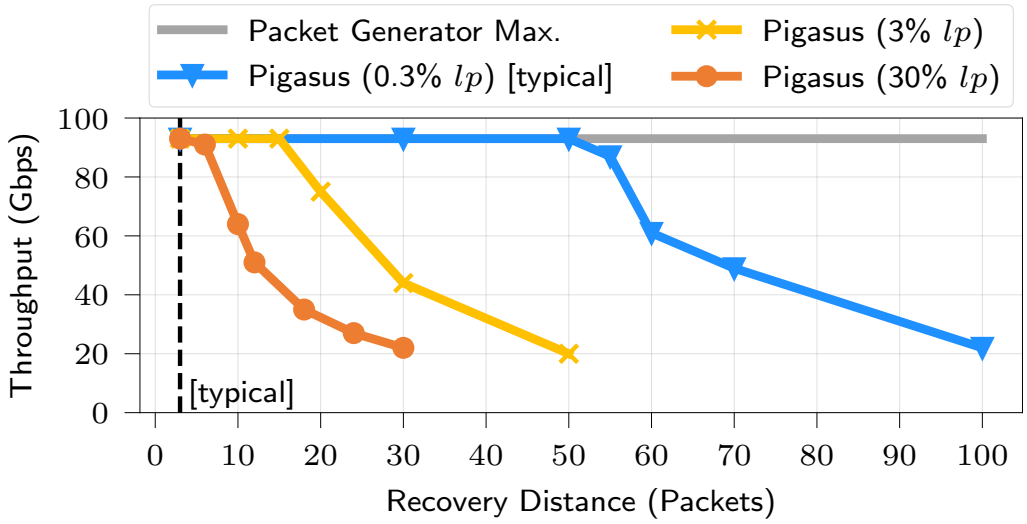


Figure 5.6: Zero-loss throughput achieved by Pigasus for a range of Loss Probabilities (lp) and Recovery Distances (rd).

empirically, the OOO path is underutilized). Since our BRAM is dual-ported, we allow the Fast Path direct access to the Flow Table, while accesses originating from the OOO Engine or Insertion Engine are managed by an arbiter that enforces the aforementioned priority scheme.

5.4 Evaluation

Setup: We evaluate our fast-slow path SRAM-based TCP Reassembly design from two aspects. First, we conduct a limit study where we stress our design using highly out-of-order packets, as we observed experimentally that the slow path is mostly idle when running over our empirical traces. Second, we compare the memory consumption of our design and the static fixed-size buffer design.

Limit study of out-of-order degree: We characterize the OOO degree using randomly generated synthetic packet traces controlled by two independent variables:

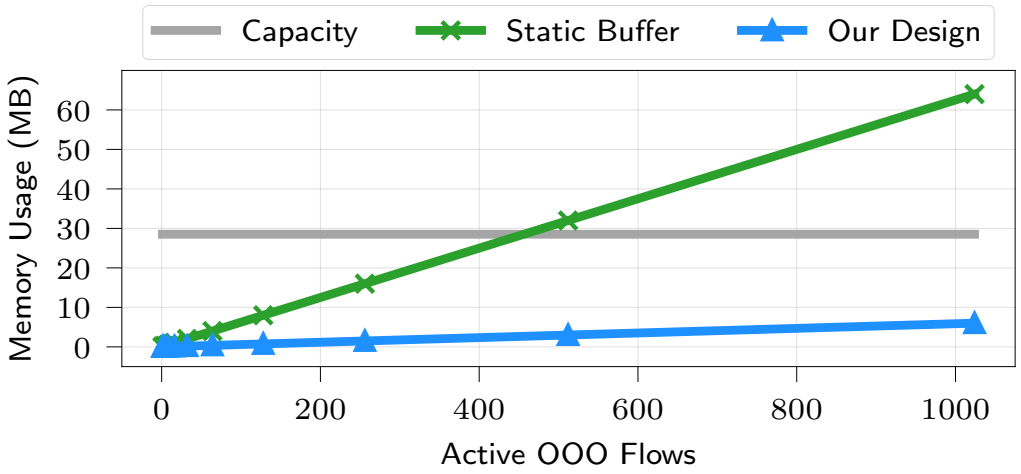


Figure 5.7: Packet buffer memory usage of different designs.

the packet loss probability (lp) [58] and the recovery distance (rd).[4] Figure 5.6 depicts the impact of these parameters on Pigasus' end-to-end, zero-loss throughput. We sweep the loss probability from 0.3% to 30%, and the recovery distance from 3 to 100. At typical values ($lp = 0.3\%$, $rd = 3$), Pigasus achieves a throughput of 100Gbps, which degrades gradually with increasing packet loss and recovery distance. It is worthwhile to note that, at typical packet loss rates, the Reassembler can handle around 50 OOO packet arrivals without any degradation in end-to-end throughput.

Memory consumption comparison with static buffer design: We plot the packet buffer memory consumption of different designs when supporting different numbers of 'Active OOO Flows' in Figure 5.7. 'Active OOO Flows' refers to the number of OOO flows that are active at the same time. This number determines the

[4]Recovery distance is defined as the number of same-flow packets that arrive before a hole created by a lost packet is filled. In Pigasus, this value determines the amount of work (in cycles) that the OOO Engine must perform for each packet that arrives out-of-order.

minimal packet buffer size before it overflows. If we follow the 'Static Buffer' design, we will need 64KB for each of the OOO flows, exceeding the 30MB on-chip SRAM (BRAM + eSRAM) capacity of our board with only 400 active OOO flows. Since our target is 100K active flows, and about 1% of the active flows are OOO, 'Static Buffer' obviously does not meet our requirement. Our design essentially allocates the buffer space on-demand, with on average 6KB for each active OOO flow based on our profile of the empirical traces. Therefore, we can support 100K flows (1K OOO active flows) only using about 6MB of eSRAM.

Chapter 6

Hierarchical Pattern Matching

Checking tens of thousands of string patterns against a 100Gbps bytestream makes the Multi-String Pattern Match (MSPM) module by far the most operation-intensive and performance-critical component in Pigasus. This chapter presents how we apply our common-case tuning insight to MSPM, yielding 100Gbps performance, only using 3.3MB of BRAM. Specifically, we use compact frontend filters to scan the common case (all packets with all rules), which allows us to use fewer replications of expensive backend filters to scan the uncommon case (suspicious packets with partially matched rules). The chapter is organized as follows: we first describe the role of MSPM and discuss why prior work does not work for us; then, we present our hierarchical MSPM design and evaluation.

6.1 Role of Multi-String Pattern Matching

As explained in Section 2.1, a Snort rule comprises three classes of patterns: a *header match*, a set of *exact match strings*, and a set of *regular expressions*. A packet triggers the rule iff *all* patterns are identified.

To avoid checking every single pattern for every index and every packet, rulesets are designed for a two-step matching process. In Snort, the MSPM is responsible for checking *header matches* and *one, highly-selective exact match string*, called the *fast pattern*. Only packets which match both the header and the fast pattern are forwarded to the Full Match stage, which checks regular expressions and any secondary exact match strings (referred to as *non-fast string patterns*). Pigasus' MSPM checks for fast patterns, headers, *and* non-fast patterns, reducing the load of Full Match on CPU.

6.2 Multi-String Pattern Matching Design Landscape

To the best of our knowledge, there are no other hardware or software projects reporting multi-string matching of tens of thousands of strings at 100Gbps. In this section, we describe the classic state-machine-based FPGA design and the Hash Table-based software design – Hyperscan – which inspires our design.

State-machine-based FPGA design: Classic FPGA-based designs adopt a state-machine approach, more specifically, Deterministic Finite Automaton (DFA) [59, 60] or Nondeterministic Finite Automaton (NFA) [19]. In these designs, the multiple string patterns are constructed as a state-machine; each byte[1] of the input stream will invoke a state transition. A rule will be matched when the input bytestream exercises a certain state transition path and reaches the final state of the rule. Multiple rules can be matched simultaneously, as rules with the same substrings may share the same set of states. A nice property of this approach is that it is *input-independent* – the system can always accept a fixed number of bytes per cycle, no

[1]Some work supports multi-striding – checking multiple bytes per cycle at the cost of decreased clock frequency and exponential growth of resource utilization [19, 61].

matter how many rules these bytes trigger. In other words, the performance does not depend on the content of the bytes.

However, state-machine-based designs are very memory/logic-consuming, and hence only work for up to a few hundreds of rules. The state-of-the-art NFA-based (already cheaper than its DFA counterpart) work [19] would require 23MB of BRAM to represent the exact match search strings alone (ignoring the additional header matches), which again exceeds the capacity of our board.

Snort 3.0 + hyperscan software design: In Snort 3.0, the MSPM is implemented using Intel's Hyperscan, illustrated in Figure 6.1.

Packets are first checked for their *header match*. Across all 10K rules, there are only ≈400 *unique* header match values. Rules which share the same header match fields are said to belong to the same *port group*. The port group module outputs a set of *port group IDs* which the packet matches; this output set is never empty, because some rules wildcard their header match and hence match all packets. An average packet matches 2 port groups.

Packets are then checked for their *fast pattern string match*. For each port group, there exists a *string matcher* which checks fast patterns for all rules within that port group. Snort must check every string matcher for each port group the packet matches.

Within the string matcher, Snort must iterate over every index of the payload, checking whether it matches any of the fast patterns in the port group. Rather than using a state machine to do this, Hyperscan uses a collection of Hash Tables. For each possible fast pattern length[2] a Hash Table is instantiated containing the fast patterns of that length. Hyperscan then performs an exact-match lookup for

[2]Up to 8 bytes – longer fast patterns are truncated.

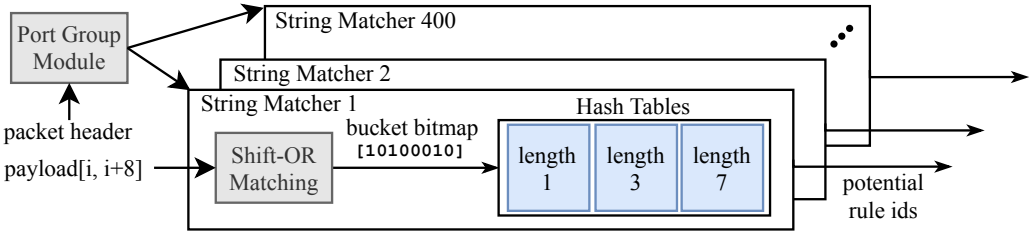


Figure 6.1: MSPM in Snort 3.0. Every String Matcher selected by the Port Group Module is evaluated sequentially.

all substrings at each index, looking up whether or not the substring is in the Hash Table – potentially $(8 \times l)$ lookups for a packet of length l.

To reduce the number of expensive sequential lookups, each string matcher contains a SIMD-optimized *Shift-Or filter* [62] prior to the Hash Table; this filter outputs either a '0' or '1' for every byte index of the packet, indicating whether or not that index matches *any* fast pattern in the Hash Tables; indices which result in a '0' output from the Shift-Or stage need not be checked. The leverage of SIMD instructions contributes most to the substantial speedup over the state machine approach.

The string matcher – combining Shift-Or and Hash Tables – then outputs a set of *rules* which the packet matched both in terms of *header* and *fast pattern*; together, the packet and the potential rule matches are passed to the full matcher. However, for most packets, this stage outputs the empty set and the packet bypasses the full match stage entirely.

6.3 Hierarchical Multi-String Pattern Matching

A straightforward port of the Snort 3.0 MSPM engines and data structures onto the FPGA consumes 785KB of memory and forwards at a rate of 3.2Gbps. Taking ad-

vantage of the high degree of parallelism offered by the FPGA, one could, in theory, scale to 100Gbps via *data parallelism*, *i.e.*, replicating this 32 times. Unfortunately, doing so would require 25MB of BRAM. We now describe how Pigasus re-architects the Hyperscan algorithm to achieve this high degree of parallelism within available resources. Since this results in leftover memory, we can then *extend* Pigasus' MSPM to scan for non-fast string patterns as well.

As shown in Figure 6.2, Pigasus flips the order of Hyperscan's MSPM, starting with string matching before moving on to header matching (port grouping).

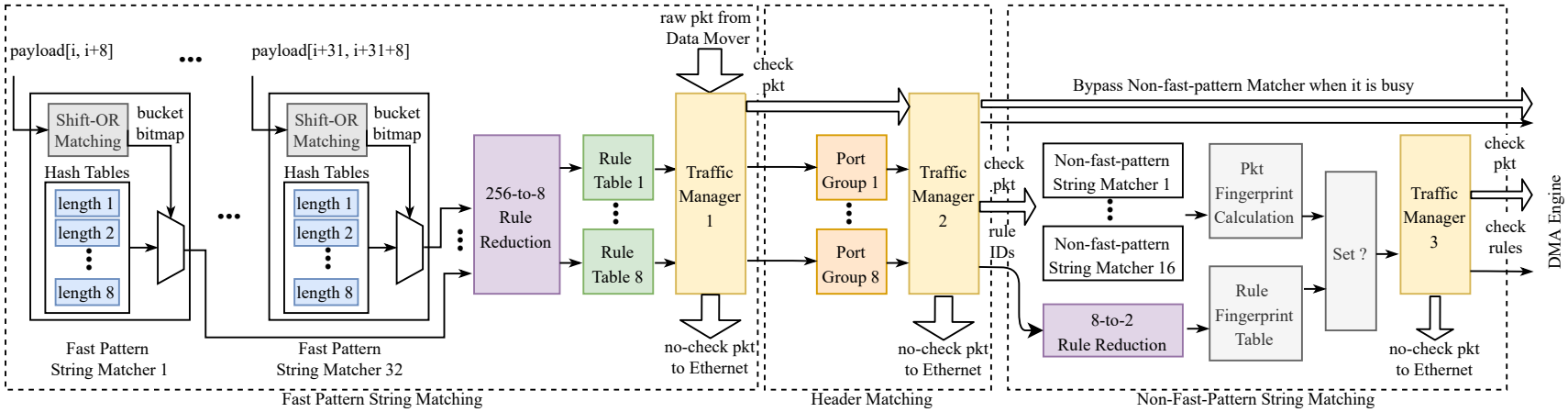


Figure 6.2: Pigasus' MSPM, which requires a total of 3.3MB of BRAM

6.3.1 Fast Pattern String Matching (FPSM)

To perform string matching, Pigasus (like Hyperscan) also has a *filtering* stage in which packets traverse two parallel filters: a shift-or (borrowed from Hyperscan) and a set of per-fast-pattern-length Hash Tables. We check the Shift-Or and (32×8) Hash Tables in parallel. Hash Tables only store 1-bit values indicating whether a given (index, length) tuple results in a match – but they do not store the 16-bit rule ID to save memory consumption. The rule ID can be inferred from the (index, length) tuple in the end of this FPSM stage, as we will discuss later in this section. The output from the filters is ANDed together, reducing *false positives* from either filter alone by 5×.

The Shift-Or and 1-bit Hash Table[3] consume only 65KB and 25KB respectively, thus they are relatively *cheap* to replicate 32× over in order to scale to 100Gbps. In theory, these filters can generate (32×8) matches per cycle (*i.e.*, 8 matches per filter); however, in the common case, most packets and most indices *do not match any rules*, and therefore require *no further processing*.[4] This gives us the opportunity to make subsequent pipeline stages narrower.

We design a 'Rule Reduction' module that selects non-zero rule matches from the filter's 256-bit wide vector output and narrows it down to 8 values. As shown in Figure 6.3, the Rule Reduction logic is a binary tree of arbiters. However, we observed empirically that some traces may generate a burst of rule matches, making the Rule Reduction logic the performance bottleneck as each of the arbiters can only serve one valid input at one time. Further investigation reveals that most of the rules are redundant spatially (same rule generated across different byte positions)

[3]Subtly, this is not a true Bloom filter [63, 64] because we only perform one hash per input; implementing multiple hashes increases resource utilization and complexity, we find, with little gain.

[4]Note that our filters never produce false negatives.

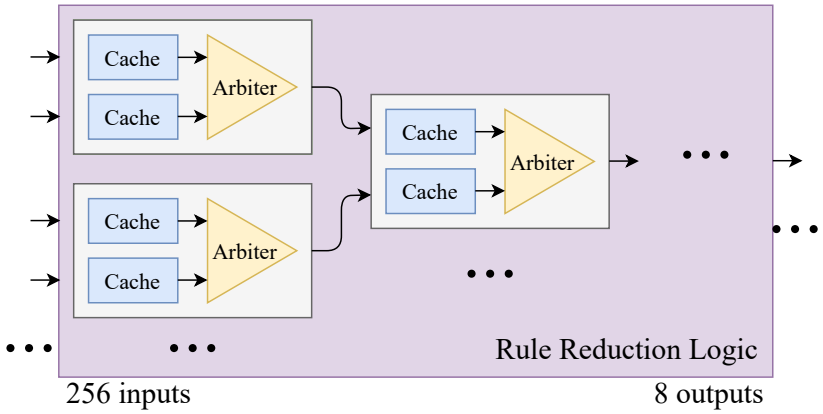


Figure 6.3: Rule reduction logic

and temporally (same rule generated across different cycles). We add a small cache (one entry) in front of the arbiter storing the last rule to reduce the load temporally. The adjacent rule reduction units also merge their outputs if they are the same to reduce the load spatially. Finally, we insert FIFOs between each reduction unit to provide necessary elasticity. These shallow (32 or 64 depth) FIFOs are implemented in LUTRAMs without wasting BRAMs, as BRAMs have minimal depth of 512.

After Rule Reduction, we only need to create 8 replicas of the 17KB Rule Table. Using the (index, length) tuple that resulted in a match in the FPSM stage, we look up the corresponding rule ID in the Rule Table. If a packet does not trigger any rules, then it will be forwarded to Ethernet without entering the next filter. As we will show in Section 6.4, with only 1.4MB BRAM consumption, this stage is able to filter out 45% of packets and, more importantly, reduce the rule matching work from *10K* rules-per-packet to *3.9* rules-per-packet for the following stages.

Applying this filter first allows us to use fewer replicas of subsequent data structures (which are larger and more expensive), since most bytestream indices have already been filtered out by the string matcher. This enables high (effective)

parallelism with a lower memory overhead.

6.3.2 Header Matching (HM)

In this stage, we use the packet header data to determine whether the matches produced by the previous (FPSM) stage are consistent with the corresponding rule's Port Group. Though at this point we only need to check 8 rules simultaneously, if not designed carefully, the header matching logic could also be a memory or performance bottleneck. The challenges come from the variety of matching scenarios. For instance, a rule can belong to a variable number of Port Groups; each Port Group can have a single port value, or a variable number of port ranges, or a variable size list. We address the problem using a combination of *specialization* and *reasonable worst-case allocation* to achieve input-independent performance with only 68KB for each rule check unit.

Figure 6.4 illustrates our header matching design. Each *Rule_Unit* matches the packet's *(protocol, source port, destination port)* with one *rule_ID*. To catch up with the hit rate of the previous FPSM filter stage, we need to replicate the *Rule_Unit* 8 times. Within each *Rule_Unit*, the *rule_ID* is used to index the *Rule2PG* table which returns up to 4 *PortGroup_IDs* since one rule can belong to at most 4 Port Groups. These *PortGroup_IDs* are used to select the relevant *PG_Unit*. Inside each *PG_Unit*, the *PG_Action* returns an action – looking up a particular table. Here, we specialize different cases (*e.g.*, single port, ranges of ports, *etc.*) into separate tables with different sizes and different types of memory (*e.g.*, the *Range* and *List* tables are very shallow, hence we map them to LUTRAM to avoid wasting BRAM space). Moreover, we treat the HTTP Port Group specially, as it is the only case that contains about 100 ports. After *specialization*, then it is reasonable to allocate

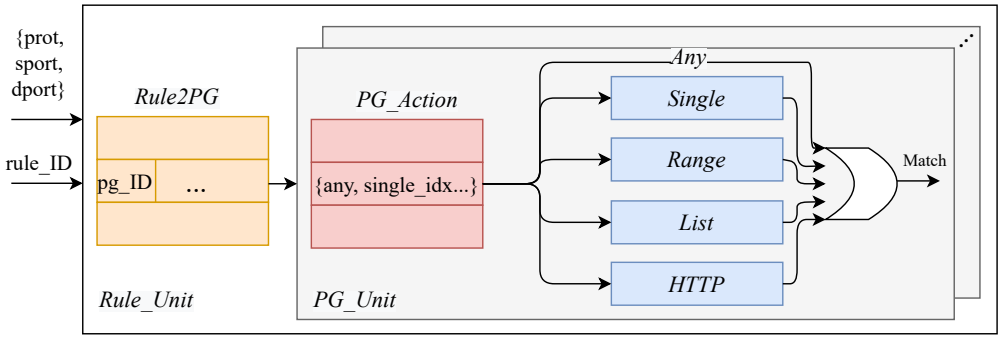


Figure 6.4: Header matching design

for *worst case* to make the design fully pipelined, since the worst case now becomes cheap enough. For example, the rest of the Port Groups only have up to 11 ports in List and up to 4 different Ranges. In the end, all of the parallel table lookup results are merged to generate the final match result.

Our initial design of the MSPM stopped here (at the Traffic Manager 2 stage in Figure 6.2), aiming merely to reproduce Snort's functionality, which only scans for fast patterns and headers. Packets which matched the fast pattern and header on at least one rule were sent to the CPU for processing, while packets which did not produce any matches at either the FPSM or Header Matching stage were simply streamed to the output interface.

This stage reduces the packet percentage from 45% to 7% with 3.2 rules-per-packet using only 0.5MB of memory. Given that this amounted to a fraction of our resource budget for the MSPM, we asked ourselves: can we do more?

6.3.3 Non-Fast Pattern String Matching (NFPSM)

Pigasus further filters down the packets and rules destined for the CPU to only 4.6% of packets, with just 2.6 rules/packet (on average), by additionally searching for *all* string matches within a rule on the board. Note that, on average, only 7% of packets

reach the Non-Fast Pattern Matcher, and, by this point, we know which rules (on average 3.2 of them) the packet might match on. Naïvely, one might iteratively search for each string in the ≈ 3.2 rules, but because each packet has a variable number of rules and each rule has a variable number of non-fast string patterns (between 1 and 32) to check, this approach would likely lead to low throughput and/or pipeline stalls.

Instead, Pigasus once again uses a set of Hash Tables (like in the FPSM) to search for all strings simultaneously. It then creates a compact, bloom-filter-like representation ('fingerprint') of the matched strings as shown in Figure 6.5. To compute the fingerprint, we first represent the set of (index, length) tuples generated by the 8 NFPSM Hash Tables as a 16-bit vector by setting $bit[index \mod 16]$ to '1' for each length bucket. Next, for each bucket, all of the 16-bit vectors generated for a given packet are ORed together to create a 16-bit 'sub-fingerprint' for that bucket. Finally, these sub-fingerprints are concatenated into a 128-bit fingerprint representing the entire packet.

The NFPSM can now look up a corresponding fingerprint – generated in the same way – for each of the ≈ 3.2 rules, and can do a parallel set comparison between the two fingerprints. If, for every bit in the *rule's fingerprint*, the corresponding bit in the *packet's fingerprint* is also set, there is a high probability that all of the exact match strings for the rule would match. But, if any of the corresponding bits are *not* set, we can be certain that at least one of the non-fast pattern strings were not matched, thus eliminating the rule as a potential match. The 4.6% of packets which match at least one rule fingerprint are forwarded to the CPU; the remainder are released as non-matching and therefore innocent packets.

It is worth noting that, as the last stage of our hierarchical filtering, the non-

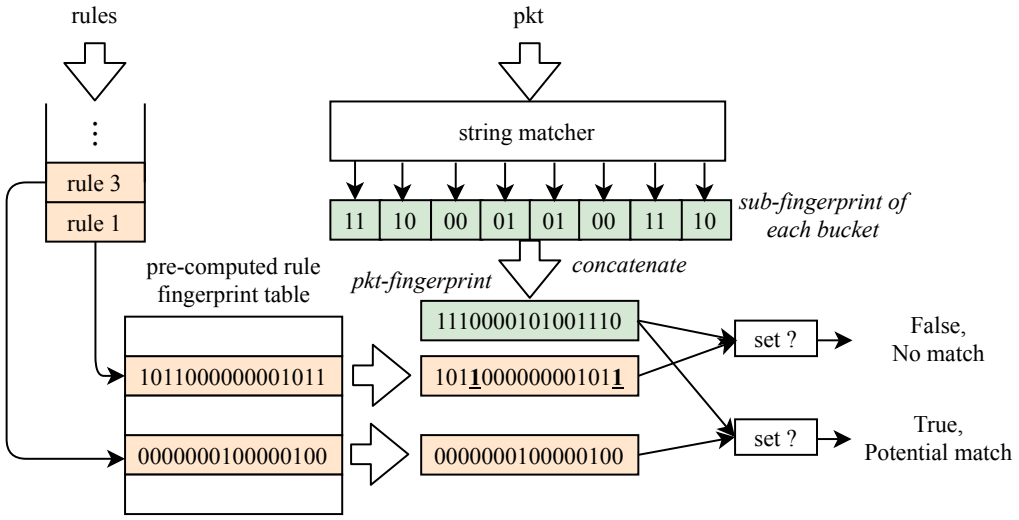


Figure 6.5: Rule matching fingerprints in the NFPSM

fast pattern matcher has the lowest throughput capacity. This saves on resources, but can make the NFPSM vulnerable to overload. In Chapter 8, we introduce the dynamic spillover mechanism to address this problem.

6.3.4 Discussion

By hierarchically filtering out packets, the MSPM reduces the amount of traffic traversing each subsequent stage of the MSPM. This means that the earliest stages require high levels of replication, but the latter stages can, on average, expect lower throughput and hence require less replication. Consequently, latter stages require lower memory consumption. End-to-end, the MSPM requires 3.3MB of memory, fitting well within our BRAM bounds while doing *more* filtering than what a naïve port of the Hyperscan algorithm would be capable of.

Although Hyperscan and Pigasus use the same set of datastructures, their rearrangements reflect fundamental differences in how one optimizes software versus

hardware.

Hyperscan first targets fitting table lookups *in cache*: this is the reason Hash Table lookups are partitioned in port groups. In contrast, on the FPGA, we have careful control of what data sits in fast BRAM versus slower eSRAM or DRAM – thus, in the hardware design, *all memory lookups take only one cycle*. Secondary to fitting in cache, Hyperscan aims to do *as few table lookups as possible*. In Pigasus, on the other hand, any compute that can be easily parallelized is cheap – and hence the first round of our filter does *256 Hash Table lookups in one cycle*.

Pigasus' primary objective, instead, is to fit in as little memory as possible. Where Hyperscan can assume that caching and prefetching will provide a reasonable illusion of infinite, processor-local memory, Pigasus must *in practice* fit all of this data in BRAM and thus structures its Hash Tables to be memory-minimal in aggregate.

6.4 Evaluation

Setup: We first evaluate the effectiveness of each filter layer by plotting the percentage of packets that are matched and the number of rules matched per packet. We then compare the memory consumption with other design options. Finally, we list the resource breakdown of each filter and its sub-modules. The setup is same as Section 3.3.1.

Match packet ratio of each filter: Figure 6.6 shows the percentage of total packets that are matched by each filter. In other words, this is the percentage of packets which are not filtered out by the current filter and thus enters the next stage. As we can see, the multiple orthogonal filters are very effective – the geometric mean

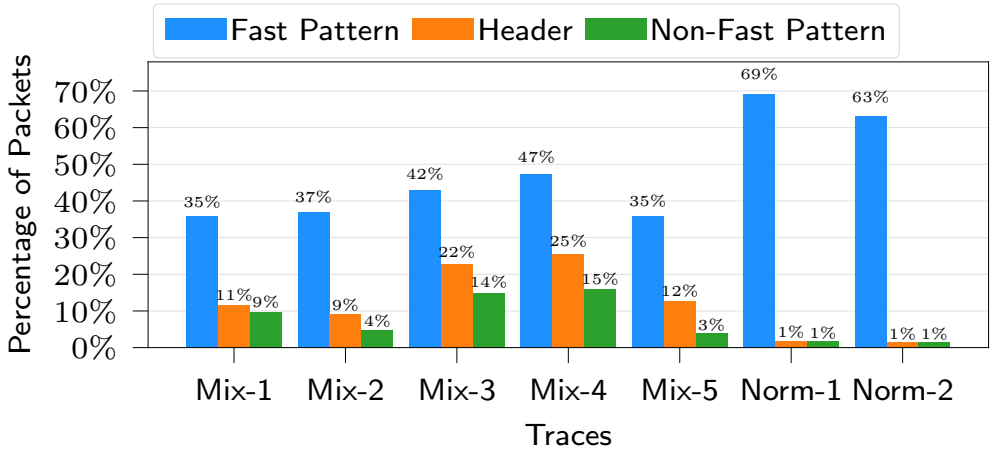


Figure 6.6: Match packet ratio of each filter stage

of packets that leave the final stage is only 4.6%. This is only achieved by applying all three filters. For instance, if we just apply Fast Pattern String Matching, about 45% of the packets will enter the CPU. Furthermore, different traces show diverse behaviors. For example, mix-3 and mix-4 have a high percentage of packets (14.9% and 15.9%) that go to the CPU. But norm-1 and norm-2 have a very high fast pattern matching ratio but a low header matching ratio. Further investigation of norm-1 and norm-2 traces reveals that some TCP packets trigger a particular UDP rule very often at the Fast Pattern Matching stage, but get rejected by the Header Matching due to protocol mismatch.

Figure 6.7 shows the average number of rules per packet that leave each filter. In the end, we only have 2.6 rules-per-packet[5] for the CPU to check. The FPSM effectively reduces the work from 10K rules-per-packet to on average 3.9 rules-per-packet for latter stages, and HM further reduces to 3.2 rules-per-packet. Note that

[5]This value is slightly higher than what we reported in our OSDI paper [51]. This is because in the OSDI paper we used an approximate metric to estimate the rules (*i.e.*, rule flit, pack of multiple and variable number of rules), but here we count the exact number of rules.

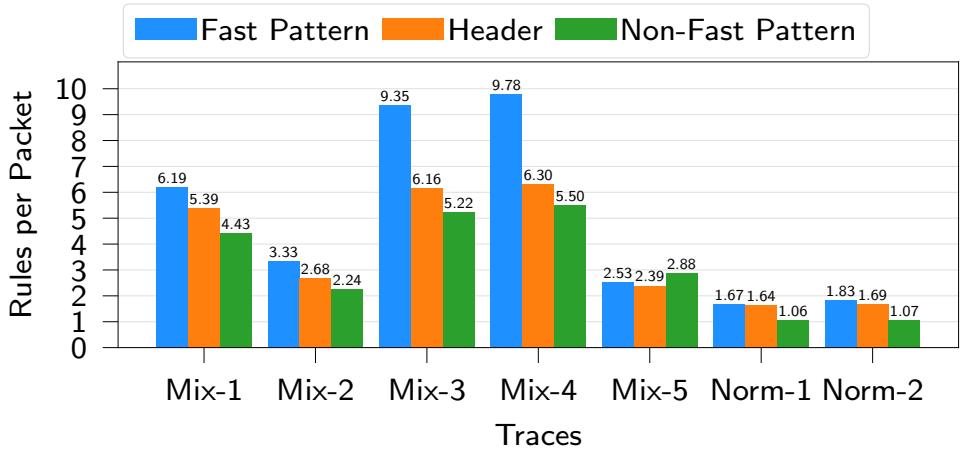


Figure 6.7: Number of rules per packet of each filter stage

rules-per-packet is a normalized value, which does not always reduce as we pass a filter. For example, in mix-5, the rules-per-packet of NFPSM is slightly higher than HM. This is possible as NFPSM may filter out the packets with fewer rules, and thus the leftover packets have on average more rules.

Resource consumption: Our hierarchical MSPM uses 3.3MB BRAM in total, an order of magnitude smaller than the NFA-based solution, which would take 23MB BRAM (only for fast pattern matching) and a straightforward input-independent implementation of a Hash Table-based solution which would consume 25MB BRAM (only for fast patterns and header matching).

Table 6.1 shows the resource breakdown of MSPM and each filter stage. The configurations are: 32 Bytes-per-cycle FPSM, 8 rules-per-cycle HM, 16 Bytes-per-cycle and 2 rules-per-cycle NFPSM. We note that we fully utilize the two ports of BRAM block whenever possible to save memory. For example, the FPSM Shift-Or instances only have 16 copies, as each two bytes can access the two ports of the

Module	ALM	BRAM (MB)
FPSM	**106,790 (15.2%)**	**1.57 (9.41%)**
Shift-Or	694 (0.1%)	1.02 (6.08%)
Hashtable	40,962 (5.83%)	0.39 (2.34%)
Reduction	27317 (3.89%)	0 (0%)
Rule_table	320 (0.05%)	0.17 (0.99%)
HM	**12,601 (1.79%)**	**0.54 (3.21%)**
Rule2PG_table	200 (0.03%)	0.15 (0.88%)
PG_Action_table	0 (0%)	0.04 (0.23%)
Single_table	0 (0%)	0.04 (0.23%)
Range_table	1,920 (0.27%)	0 (0%)
List_table	1,920 (0.27%)	0 (0%)
HTTP_table	0 (0%)	0.31 (1.87%)
NFPSM	**23,324 (3.32%)**	**1.25 (7.51%)**
Shift-Or	1,712 (0.24%)	0.88 (5.26%)
Hashtable	13,252 (1.89%)	0.21 (1.29%)
Reduction	2,114 (0.30%)	0 (0%)
Fingerprint_matcher	800 (0.11%)	0.16 (0.96%)
MSPM	**142715 (20.31%)**	**3.36 (20.13%)**

Table 6.1: Resource breakdown of MSPM and each filter. Percentages are relative to the total amount of resources in a Stratix 10 MX FPGA.

same table simultaneously.

Discussion: Our MSPM design again adopts our common-case tuning insight, separating the common case (all packets and all rules) from uncommon case (a few suspicious packets and partially matched rules). This separation allows us to achieve 100Gbps overall throughput using only 3.3MB of BRAM for the empirical traces we evaluated.

However, from Figure 6.6, one may already notice the diverse behaviors among traces, suggesting that there is no 'one-size-fits-all' configuration, as a single design can either cause performance degradation or waste resources. For instance, for norm-1 and norm-2, adding a non-fast pattern filter does not help the performance that much. However, if we remove this filter, the mix-1 to mix-5 traces will

suffer from low performance. Therefore, users should be able to easily re-tune the configuration at both compile time and runtime to efficiently adapt to new environments. Nevertheless, this is challenging as our initial design follows the traditional static 'ASIC-style' design mindset, which assumes that once the implementation is deployed it rarely changes. Combined with our common-case tuning insight, our initial design is brittle to changing environments. In Chapter 7 and Chapter 8, we will show how we address this problem at compile time and runtime respectively.

Chapter 7

Disaggregated Service-Oriented Streaming Design

In previous chapters (Chapters 4, 5, and 6), we focused on our first insight – *tune for common case*, where we proposed multiple new ideas to improve the common-case performance at different levels of the system, while minimizing the resource consumption for a set of *given inputs*. While these specializations are crucial for high efficiency, they can also lead to overfitting – the design works well for one trace but may not be ideal for others (as discussed in Section 6.4). This is undesirable given that a real-world deployment could be much more diverse than the set of empirical traces we evaluated.

To address the overfitting problem caused by the common-case specialization, we propose a *disaggregated and dynamic architecture*. In particular, we will focus on the disaggregation in this chapter and defer the dynamic adaptation to the next chapter (Chapter 8). The key difference between the disaggregated service-oriented streaming design of Pigasus (*Pigasus 2.0*) and the initial 'ASIC-style' Pigasus design

(*Pigasus 1.0*) is that Pigasus 1.0 is a *one-shot design* specialized for a single input rate (100Gbps), a particular FPGA (Intel Stratix 10 MX), and specific empirical traces (Stratosphere [46]), while Pigasus 2.0 is a *design template* that allows users to conveniently recompose a library of disaggregated and parameterizable streaming services through a common infrastructure abstraction to achieve best efficiency in various deployment environments. The remainder of this chapter is organized as follows: we first motivate the problem by showing the limitations of Pigasus 1.0 that follows the traditional design methodology and discuss related work in this space; we then present Pigasus 2.0, a disaggregated service-oriented streaming design template, and its use cases; finally, we evaluate the performance, resource utilization, scalability, and portability of Pigasus 2.0.

7.1 Motivation

Pigasus 1.0 follows the traditional 'ASIC-style' design methodology [43], where a design rarely changes once deployed. In this methodology, developers commit to 'one-size-fits-all' – a single design needs to work for all possible inputs, which is probably fine for *input-independent* applications, but not for *input-dependent* applications such as IDS/IPS. As we have already shown in Section 6.4, the behaviors of different traces vary a lot from each other, *i.e.*, NFPSM is very effective for mix-2 to mix-5 traces, but not for norm-1 and norm-2. This suggests that for some inputs, it might be more resource efficient to discard NFPSM completely with little performance impact. Another possible scenario is that some stages (*e.g.*, Header Match) are triggered more often for particular inputs, requiring scaling up/out of that stage to keep up with the target performance.

With Pigasus 1.0, there are two possible options to handle the aforementioned

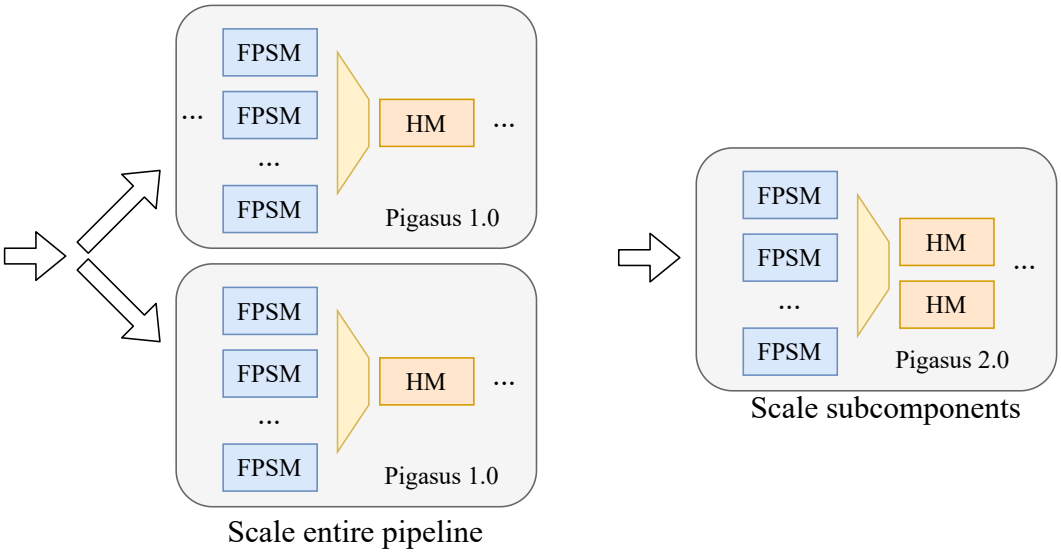


Figure 7.1: Scaling Pigasus as the Header Matcher becomes the bottleneck. Pigasus 1.0 has to scale the entire pipeline while Pigasus 2.0 only needs to scale its subcomponents.

scenarios: (1) duplicating the entire pipeline, and (2) modifying the internal designs in Pigasus. While duplicating the Pigasus 1.0 pipeline entirely and distributing traffic among the two pipelines can address the performance bottleneck of the Header Match stage as illustrated in Figure 7.1, it is clearly not as resource efficient as scaling up/out just the bottleneck stage itself. On the other hand, since the internal modules are tightly coupled for best efficiency in Pigasus 1.0, it is labor-intensive and error-prone for users to manually change the RTL codebase.

We were initially motivated by just handling different inputs efficiently, but here we instead ask a more general question – how should we design Pigasus without prior knowledge of the deployment environment (including the inputs, target FPGA, and target performance), while allowing users to easily re-tune the design at compile time for high efficiency.

7.2 Related Work

Before we dive into the details of our design, we first present related efforts, some of which served as inspiration for our design.

Hierarchical modular design practice: This is a common design practice for developing a large complex system in RTL. We argue that it is necessary, but not sufficient for our purpose. In particular, it misses (1) interface standardization which allows easy management at the system-level even though it may sacrifice some efficiency due to the generalization (*e.g.*, packing/unpacking, encoding/decoding, *etc.*); (2) parameterization that enables easy scaling up or scaling down of certain modules without changing the RTL code; and (3) automatic system composition that automates the tedious, error-prone process of recomposing modules whenever a design changes.

IP developments: Intellectual Property (IP) based hardware design shares the same insight with the 'library' in software development, wherein a well-encapsulated functionality can be easily reused for many different designs. Today's IP development provides standard interfaces with parameterization. The Smart IP concept [65] even encapsulates the IP authors' domain knowledge for better debuggability and parameter selection. Inspired by IP development, we argue that application developers should disaggregate a design and develop each building block as an IP, instead of following the 'ASIC-style' design – creating a one-shot, highly customized implementation. In this way, at compile time, users can parameterize and compose those IPs to best suit the new environment.

Design abstraction: This line of work also targets easy development but from a different aspect of IP development. For instance, service-oriented architecture [66]

decomposes the memory access operations from the processing kernel and forms a 'memory service' which raises the memory access abstraction from simple loads/stores to 'objects,' *e.g.*, adjacent node in graph, next image, *etc.* More importantly, this work separates the 'logical design' from its 'physical implementation,' allowing the same design to be easily implemented differently as the performance/resource constraints or platforms change. This inspired us to propose a common communication abstraction for streaming applications for better portability and improved design productivity.

7.3 Pigasus 2.0 Design

Our objective is to develop Pigasus in a way that it can be easily re-tuned for high efficiency when the deployment environment – the inputs, the target FPGA, and even the target performance – is different from our initial setting. To achieve this goal, instead of providing *a single design*, we should provide *a space of designs* and let users pick the most appropriate configuration for best efficiency in the new deployment environment. Following this idea, we developed Pigasus 2.0, as shown in Figure 7.2.

Pigasus 2.0 is a set of disaggregated, parameterizable services that allows users to conveniently scale up/down certain building blocks to adapt for a new environment at compile time. The common communication abstraction for streaming services further improves the productivity and portability by providing a rich set of features and abstracting the platform-specific interfaces such as Ethernet, PCIe vendor IPs, *etc.* Below, we describe the three key pieces of our solution: *disaggregation*, *parameterization*, and *communication abstraction*.

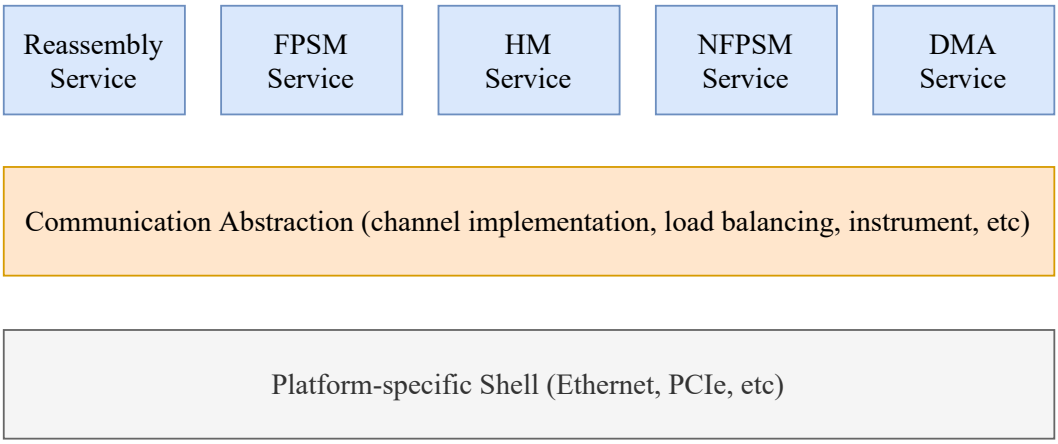


Figure 7.2: Pigasus 2.0 architecture. Streaming services are logically connected through a common communication abstraction. The RTL implementation of the communication abstraction is automatically generated at compile time.

7.3.1 Disaggregation

The root cause of the inefficient scaling in Pigasus 1.0 is that the tightly coupled design can only be scaled at the granularity of the entire pipeline. After disaggregating the design into smaller pieces, users can scale at the granularity of each IP and hence achieve the target performance using much fewer resources. The next question that comes up is to what extent should we disaggregate the design. If we disaggregate it too little, we are less likely to gain the expected efficiency. On the other hand, if we disaggregate it too much, then the overhead of standardizing the interfaces would become noticeable and the many small services will also make it harder for users to understand and reuse. To get a balanced disaggregation, we follow the three principles below.

Minimal unit for scale out: After disaggregation, each streaming service should be the *minimal unit* one would like to scale out. For instance, scaling out the entire MSPM is too expensive if only the Header Match stage is the performance

bottleneck. However, further breaking down the Header Match stage and scaling out a particular sub-table (shown in Chapter 6) does not make sense for improving the overall performance. Therefore, the Header Match stage is a good candidate. Following this principle, we break the MSPM into 3 separate services: Fast-Pattern String Match (FPSM), Header Match (HM), and Non-Fast-Pattern String Match (NFPSM).

Preserve latency-sensitive loop: To maintain the latency-insensitive streaming semantic, we should not break any latency-sensitive loops such as a memory read/response. For example, the data mover in Figure 3.1 reads the actual Ethernet packet data from the packet buffer implemented in SRAM, which takes a fixed number of cycles to respond. In this case, since the data mover is tightly coupled with the packet buffer, we encapsulate the packet buffer, data mover, and initial reassembly logic in Pigasus 1.0 into a bigger reassembly service in Pigasus 2.0.

Expose the same standard interface: The interface exposed by each service should be the *same* to allow easier management as we will show in Section 7.3.3. The interface should also follow the industry *standard* for better compatibility. For instance, we adopted the Avalon Streaming Interface [67], which can be easily adapted to other standard streaming interfaces such as the AXI Streaming Interface [68]. The code Listing 7.1 shows the interface of a service. Each service contains three channels: a packet channel for transferring packet data; a metadata channel for additional information about each packet such as packet size, 5 tuples, *etc.*; and a user data channel for passing any user-defined data, which is used for passing rules in Pigasus. For the packet and user data channels, since the data size is variable and can be bigger than 512-bit, we need out-of-band signals, *i.e.*, startofpacket, endofpacket, and empty, to track the boundary of the data block and padding bytes

at the end of the data block.

```
module example_service (
    // Input packet data
    input   logic [511:0]   in_pkt_data,
    input   logic           in_pkt_valid,
    input   logic           in_pkt_startofpacket,
    input   logic           in_pkt_endofpacket,
    input   logic [5:0]     in_pkt_empty,
    output  logic           in_pkt_ready,
    // Input metadata
    input   metadata_t      in_meta_data,
    input   logic           in_meta_valid,
    output  logic           in_meta_ready,
    // Input user data
    input   logic [511:0]   in_usr_data,
    input   logic           in_usr_valid,
    input   logic           in_usr_startofpacket,
    input   logic           in_usr_endofpacket,
    input   logic [5:0]     in_usr_empty,
    output  logic           in_usr_ready,
    // Output direction is omitted for simplicity
}
```

Listing 7.1: Service interface sample

7.3.2 Parameterization

To allow users to effortlessly reuse Pigasus in different environments, we provide a range of parameters to trade off resource with performance/capacity for different services. This is realized through a combination of *SystemVerilog parameters* and Python-based Jinja2 templates, which are used when the former are not expressive

enough. For instance, the many submodules in MSPM (*e.g.*, Hash Tables, rule reduction, *etc.*) are coded in Jinja2 templates which generate the final SystemVerilog file. To make it simple, we provide a single entry file for the user to edit; the parameters will be parsed and passed to the Jinja2 templates for code generation.

Table 7.1 shows the key parameters and their impact. Note that we currently only support discrete power-of-2 values in the ranges listed in the 'Support' column, while the pkt buffer in eSRAM only supports two configurations. The *Num of flows* parameter can trade off BRAM consumption with flow capacity. The *Pkt buffer* is unique, as we allow users to select BRAM-based implementation when eSRAM is not available on the target FPGA, making the design more portable among different FPGA platforms. *FPSM width* refers to how many bits this service can accept per cycle, which will affect the BRAM/LUT utilization and performance. *FPSM reduction ratio* refers to the rule reduction ratio of the FPSM. If the input traffic is mostly innocent, it is more efficient to select a high reduction ratio such that the following stages could be narrower. If the traffic generates lots of matches in FPSM, it is recommended to use a low reduction ratio to allocate enough resources to avoid performance degradation. *HM width* is the number of rules the Header Match stage can process per cycle, since this stage does not need to check the bytes of the packet data. *NFPSM* has parameters similar to the FPSM.

We note that some of the parameters are coupled. For example, if the FPSM reduction ratio is high, the HM width should not exceed the final number of rules produced by FPSM. Otherwise, some of the HM processing units will always be idle. In addition, the parameters have limits, which are determined by either the algorithm we used or an FPGA design practice. For instance, the 64bit minimal width of FPSM and NFPSM are decided by the string matching algorithm, which

Parameters	Impact	Support
Num of flows	BRAM;flow capacity	2048-131072
Pkt buffer	eSRAM;pkt capacity	eSRAM: 2688,5376;
	BRAM;pkt capacity	BRAM: 512-2048
FPSM width	BRAM/LUT;perf	64-512 bit
FPSM reduction ratio	LUT;perf	2-64
HM width	BRAM/LUT;perf	8-32 rules
NFPSM width	BRAM/LUT;perf	64-512 bit
NFPSM reduction ratio	LUT;perf	2-64

Table 7.1: Key parameters and their impact in Pigasus 2.0

takes at least 8B per cycle. The 512bit maximum width comes from an FPGA design practice where an even wider signal should be avoided as it may cause routing congestion and hence reduce the max frequency at which the design can run. If the 512bit version cannot meet the desired performance, one should consider scaling the service by creating a replica, instead of making the single instance even wider.

7.3.3 Communication Abstraction

A streaming application can be logically considered to be a graph where the node represents the streaming service and the edge represents the communication channel between streaming services. When implementing streaming applications on FPGA, developers have been following what we call 'wire abstraction,' where the channel is implemented as directly-connected wire or FIFO. However, such a 'wire abstraction' makes porting and design exploration a time-consuming process, as developers have to manually recompose the streaming services at the RTL level after any changes in the logical connection or physical placement of the streaming services.

To tackle this problem, we propose a 'communication abstraction'[1] inspired

[1]This work is a collaboration with Joe Melber and Siddharth Sahay. I provide the RTL implementations of the building blocks, *e.g.*, different FIFO implmentations, inter-FPGA communication, *etc.* They encapsulate the building blocks with 'service' interfaces and generate the composition

by service-oriented architecture [66], where all services are connected to a *common* abstraction logically. In our communication abstraction, users can easily explore different design options – for instance, replicating/inserting/deleting streaming services – and then recompose the services at a high level (*e.g.*, Python). The compiler will take care of the RTL implementation generation, which retains the same efficiency as manual composition. Our communication abstraction also provides important features for better portability, design productivity, and debuggability as follows:

Different channel implementations: Our communication abstraction provides different physical implementations of the inter-service communication channel. This is critical as new types of FPGA systems are emerging, such as Network-on-Chip (NoC)-based FPGAs [69] or multi-FPGA systems [70], which are more promising in flexibly handling input-dependent behaviors. Mapping Pigasus to such platforms is more challenging than mapping to other monolithic FPGAs, as it requires changing the channel implementation to intra-FPGA (NoC) or inter-FPGA (Ethernet or PCIe) communication. With the communication abstraction, users only need to select the appropriate type of inter-service channel at Python-level without worrying about the RTL details.

Load balancing: We expect scaling out a specific streaming kernel will be a common design practice for new environments. As such, our communication abstraction provides load balancing services internally, allowing users to simply create replicas without being concerned about the load distribution and merging of results.

Instrumentation: This common communication abstraction is the natural place to insert instrument probes to collect statistics as the data flows between streaming kernels. This will give users the visibility into internal states, helping them to quickly

using their Python framework.

identify performance bottlenecks or localize functionality bugs.

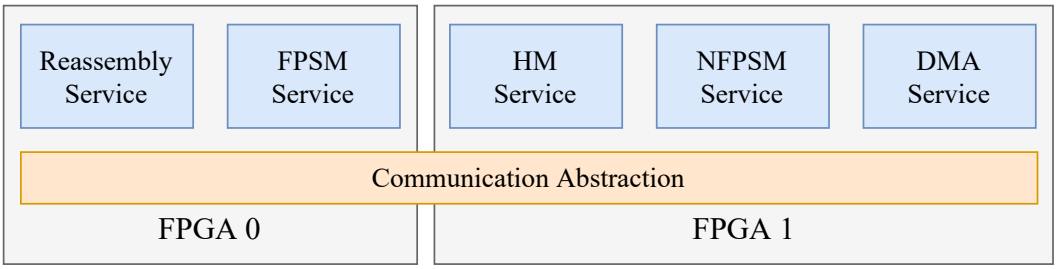
7.4 Potential Use Cases of Pigasus 2.0

By combining disaggregation, parameterazation, and the communication abstraction, Pigasus 2.0 enables easy compile-time re-tuning, which alleviates the brittleness caused by common-case tuning. In this section, we demonstrate the potential of Pigasus 2.0 by showcasing three examples.

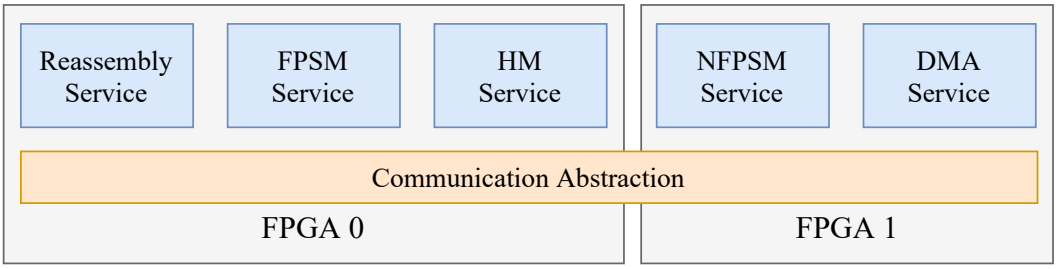
Scale up performance bottleneck: As we briefly discussed in Section 7.1, Pigasus 2.0 allows scaling up the performance bottleneck explicitly to achieve target performance with minimal resource overhead. Figure 7.1 illustrates the case when the Header Match stage becomes the performance bottleneck, where we can make it wider without replicating the entire pipeline. This scale up/out is applicable to any uncommon-case services, *e.g.*, NFPSM.

Scale down to lower line-rate: With Pigasus 1.0, if the performance target of the deployment is lower than 100Gbps, *e.g.*, 50Gbps, to use Pigasus, users have to buy an Intel Stratix 10 MX FPGA and run the 100Gbps pipeline, which is obviously not cost effective. With Pigasus 2.0, users only need to change the width parameters of FPSM, HM, and NFPSM without modifying any service-internal designs to generate a 50Gbps design point. Since the resource consumption of a 50Gbps design point would be lower than a 100Gbps pipeline, users can adopt smaller FPGAs to save cost.

Map to multi-FPGA system: With the observation that network line-rates are hitting 400Gbps, we envision that a single FPGA cannot host the entire pipeline, and thus splitting the pipeline and mapping it onto multiple FPGAs to continue



(a) Design 1, splitting at HM service.



(b) Design 2, splitting at NFPSM service.

Figure 7.3: Explore different pipeline splitting on multi-FPGA system using common communication abstraction

capacity/performance scaling becomes necessary. Pigasus 2.0 provides inter-FPGA communication as an infrastructural service, which boosts the design productivity for multi-FPGA systems – users only need a few lines of Python code change to generate the RTL implementation of different splitting points within minutes, as shown in Figure 7.3.

7.5 Evaluation

Setup: We first evaluate the performance and resource overhead introduced by disaggreation by comparing a design point of Pigaus 2.0 with Pigasus 1.0 under the same setup. We then assess the scalability of Pigasus 2.0 using two cases, scaling up a performance bottleneck and scaling down to serve a lower line-rate. Finally, we demonstrate the portability and design productivity by implementing different

multi-FPGA partitioning designs.

Performance and resources: This experiment evaluates the overhead introduced by disaggregation when compared with Pigasus 1.0, a tightly coupled, highly specialized one-shot design. The design point of Pigasus 2.0 that shares the same setting with Pigasus 1.0 does not introduce any performance degradation and only introduces 2% (of total board capacity) of LUTs and BRAM overhead. Therefore, disaggregating the design and standardizing the interfaces in Pigasus 2.0 does not lose any efficiency.

Scalability: We conduct two experiments to demonstrate that disaggregation allows more efficient scaling. In the first experiment, we make the case that Header Match is the performance bottleneck as shown in Figure 7.1, and we compare the resource consumption of 'pipeline replication' and 'bottleneck scaling.' To make the Header Match stage the performance bottleneck, we generate a semi-synthetic trace by extracting a highly matched packet from Mixed-1 trace and repeating it with an adjusted sequence number to not confuse the Reassembly stage. This semi-synthetic trace triggers many rule matches in FPSM. Since the Header Match stage is narrow, the rule reduction logic in FPSM will create pipeline stalls and slow down the entire system. With Pigasus 1.0, users have to replicate the entire pipeline in order to maintain the 100Gbps throughput. With Pigasus 2.0, users can just scale up the performance bottleneck – Header Match in this case – to let more rules go through.

Figure 7.4 depicts the resource saving between these two designs relative to the single Pigasus 1.0 pipeline. 'Pipeline Replication' will double the resource consumption. However, 'Bottleneck scaling' only introduces 10% more LUTs and 6% more BRAM to keep up with the 100Gbps line-rate for this new traffic profile.

In the second experiment, we make the case that users want to run Pigasus on

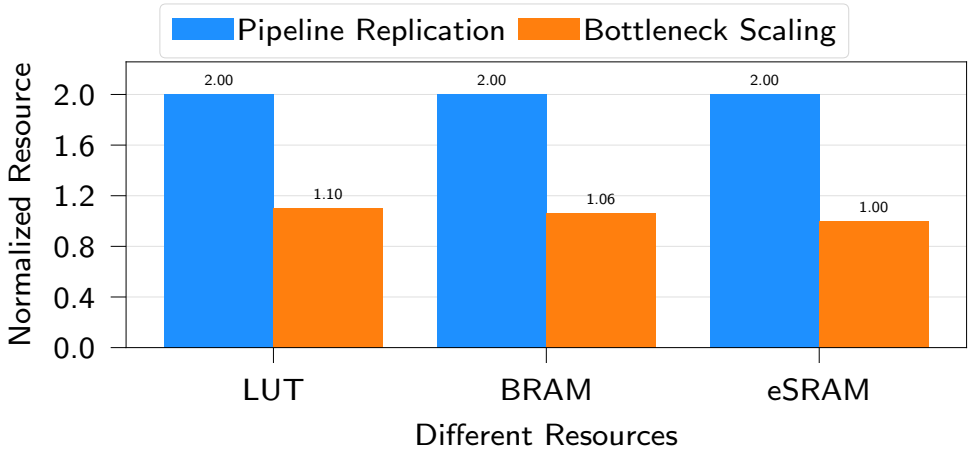


Figure 7.4: Resource consumption of pipeline replication and bottleneck scaling

a 50Gbps link, and we compare the resource consumption of the '100Gbps pipeline' with a '50Gbps pipeline.' If the performance target is 50Gbps, with Pigasus 1.0, users have to use it as is. In contrast, Pigasus 2.0 allows users to change the configurations to easily scale down performance-dependent modules to save resources.

Figure 7.5 depicts the per-module LUTs and BRAM consumption when mapped to 50Gbps. The resource consumption is also normalized to a single 100Gbps pipeline. Some of the modules are not performance-dependent, *e.g.*, Reassembly and DMA. Their resources depend on the flow capacity and packet buffer capacity, but not the throughput. The resource consumption of FPSM, HM, NFPSM, and Vendor IP depends on throughput. For instance, FPSM and NFPSM are now half the size as before, and therefore use about half the resources compared to the 100Gbps pipeline. HM is special, as it is already the minimal size (8 rules-per-cycle determined by the algorithm) in the original 100Gbps pipeline. In the 50Gbps pipeline, the resource consumption of HM remains the same. The saving in vendor IPs mostly comes from changing PCIe IP core setting from Gen3 16 lanes (theoretical peak 128Gbps) to

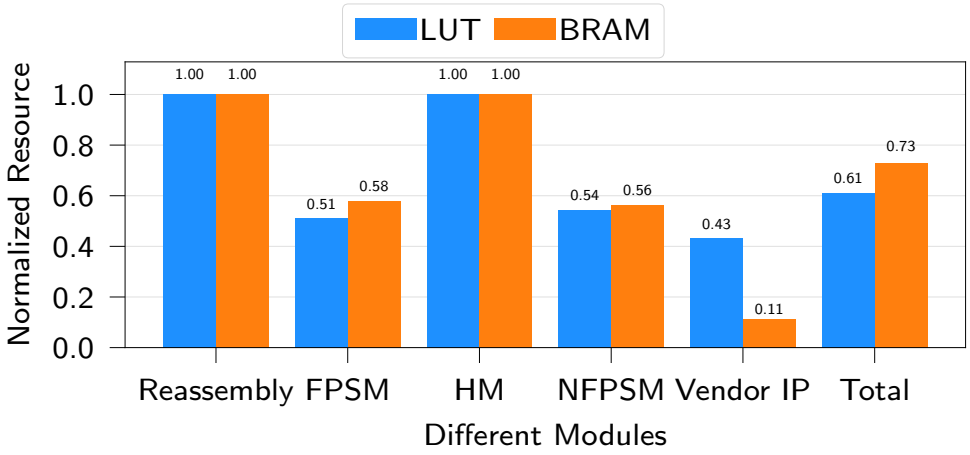


Figure 7.5: Resource consumption of 50Gbps pipeline relative to 100Gbps pipeline

Gen3 8 lanes (theoretical peak 64Gbps). Overall, the 50Gbps pipeline saves about 30% of resources compared to the 100Gbps pipeline.

Portability and design productivity: To demonstrate the portability and design productivity of the communication abstraction, we port Pigasus to a multi-FPGA system and then explore different cutting points as illustrated in Figure 7.3. In this experiment, we connect two Intel Stratix 10 MX FPGAs through another available 100Gbps Ethernet port. We chose the Mixed-5 trace for this experiment, as it shows the most interesting filtering ratios as shown in Figure 6.6.

Figure 7.6 shows the zero-loss throughput of different splitting points when normalized to the monolithic design. The splitting point means that we cut the pipeline in front of that module. As we can see, different design points have different performances. For instance, splitting in front of FPSM or HM may lead to lower performance, as the percentage of traffic that has to go across FPGAs is high and we need to pass the associated rules and metadata with the packets, which stressed

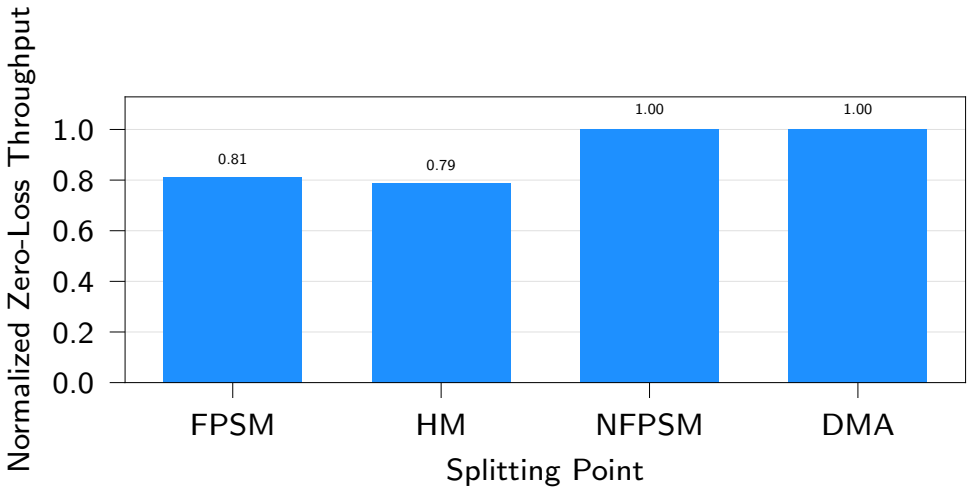


Figure 7.6: Zero-loss throughput of different splitting points normalized to monolithic design

the inter-FPGA link. For the NFPSM and DMA case, since most of the traffic is already filtered out, the inter-FPGA link is not the bottleneck anymore. These performance variances are input-dependent, suggesting that users should explore all of these options before making the decision under a new deployment environment. Pigasus 2.0's communication abstraction allows users to switch design points with only 17 lines of Python code changes and generate the RTL implementations of all of these options within minutes. This significantly improves the productivity compared to hand-crafting hundreds of lines of RTL for each design point.

Discussion: To alleviate the brittleness of the common-case tuning strategy, Pigasus 2.0 enables compile-time re-tuning for high efficiency under new deployments. In particular, disaggregation breaks the highly-specialized pipeline into individual streaming services, and parameterization provides performance/capacity/resource tradeoff for each service, enabling more efficient scaling. Since users need to tweak the parameters and try different design options, it is crucial to make the exploration

easy and fast. The common communication abstraction offers a rich set of features and can be automatically generated from a high level, significantly improving the portability and design productivity. While our compile-time re-targeting is effective and easy to use, it fundamentally falls short in handling traffic dynamism. In the next chapter (Chapter 8), we discuss how to adapt to dynamic traffic at runtime.

Chapter 8

Dynamic Spillover Mechanism

In Chapter 7, we described how to efficiently adapt Pigasus for new workloads at *compile time*. However, it is very challenging for a compile-time profile to accurately predict the behaviors of real workloads at *runtime*. As such, the design decision made at compile time is most likely to be imperfect, leading to either under-provisioning, which harms performance, or over-provisioning, which wastes resources. In this chapter, we introduce our dynamic spillover mechanism, which can efficiently adapt to changing workloads. The key idea is to dynamically bring up backup streaming services on-demand to absorb the variance of the traffic to avoid over-provisioning at compile time. In the rest of this chapter, we first discuss why existing approaches are insufficient. We then introduce our idea and discuss different design alternatives for the backup streaming services. Finally, we evaluate the effectiveness of our approach using a number of performance benchmarks.

8.1 Motivation

The problem we are facing is that input-dependent behaviors can stress different performance and resource bottlenecks in a *burst* at runtime. In other words, expensive operations may be triggered very frequently within a short period of time, causing pipeline stalls and packet drops.

A straightforward solution is to allocate enough resources for the worst case. In this way, we can safely create one implementation for deployment, which can handle all possible cases without performance degradation. However, as we have shown in Chapters 4, 5, and 6, the worst-case-oriented designs are too resource-consuming to be practical.

That is why we choose to tune for the common case, allocating just enough resources to bolster the common case performance. While tailoring for common cases enables high performance using minimal resources, it also makes the design brittle in the face of changing workloads. Taking hierarchical filters as an example, at compile time users have to decide the size of each filter based on the profile of the average filtering ratio to get a balanced and efficient design. However, at runtime, the traffic could have significantly higher variance than the profiled traces indicated; consequently, the last stage of filters might be stressed instantaneously, leading to performance degradation.

A typical way of dealing with burstiness in traffic is using buffering; however, at 100Gbps, small jitters in traffic can easily overflow our MB level of on-chip buffer space. Moreover, due to the latency requirement of real-time processing (IPS mode), one cannot buffer the traffic indefinitely.

To summarize, we cannot afford to provision for a worst-case-oriented design, and buffering does not fundamentally solve the problem. Therefore, we need a new

way to resolve the brittleness of our common-case-tailored design.

8.2 Dynamic Spillover Mechanism

Our idea is to prepare backup streaming services that can be dynamically brought up on-demand. When the packets requiring expensive processing are sufficiently spaced apart, which is the case most of the time, we only use primary streaming services. Only during periods of heavy bursts – *i.e.*, the load of the primary streaming kernel exceeds a certain threshold – the backup streaming kernel will be fired up to handle the spillover traffic until the primary streaming service's load drops below the threshold. In this way, we dynamically introduce more computing power to absorb the burstiness while maintaining low resource consumption most of the time.

This idea generalizes well to any streaming services that handle uncommon and expensive cases. Furthermore, the backup streaming services can also be implemented differently as long as they can be easily brought up. In this chapter, we demonstrate the effectiveness of this idea with one use case, where we dynamically route spillover traffic to the CPU when the NFPSM is under high load. We then discuss other potential use cases.

8.2.1 NFPSM CPU Spillover

Non-Fast Pattern String Match (NFPSM) is the last pattern matching stage of the MSPM module in Pigasus. Based on our profile of the empirical traces, we have observed that on average, 7% of packets with 1.6 rules-per-packet traffic enter the NFPSM as shown in Section 6.4. Thus, we allocate the NFPSM to be faster than the average case; for instance, our NFPSM can sustain 50Gbps of packet traffic with

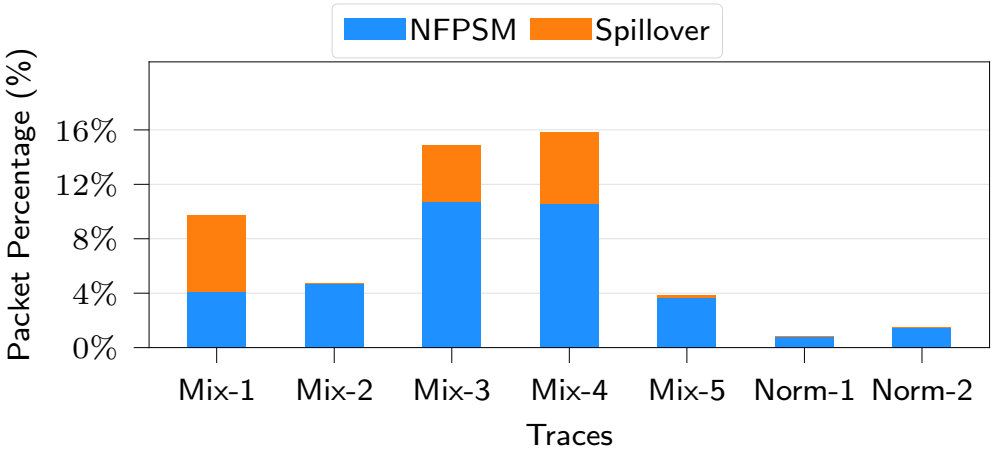


Figure 8.1: Percentage of total packets processed by NFPSM and spillover

2 rules-per-packet. Although this exceeds the *average* service requirement, bursty traffic can still overflow the NFPSM and thus stall the pipeline.

Figure 8.1 depicts the percentage of total packets that are processed by the NFPSM and the spillover mechanism among different traces. For 4 of the 7 traces, the NFPSM adequately serves the vast majority of traffic, while the spillover mechanism handles only a very small fraction ($< 1\%$) of packets. However, for mix-1, mix-3, and mix-4, noticeable amounts of traffic cannot be handled by NFPSM due to burstiness.

To address this problem, we design a spillover path that can route the overflow traffic to the CPU as shown in Figure 8.2. We create a 'Dispatcher' in front of NFPSM which monitors the load of NFPSM by continually checking the occupancy of its input buffer. If the occupancy exceeds a runtime-configurable threshold, the Dispatcher will route the spillover traffic to the DMA engine through a different path. The DMA engine will merge the data from different paths and distribute them to CPU cores. Once the load of NFPSM drops below the threshold, the Dispatcher

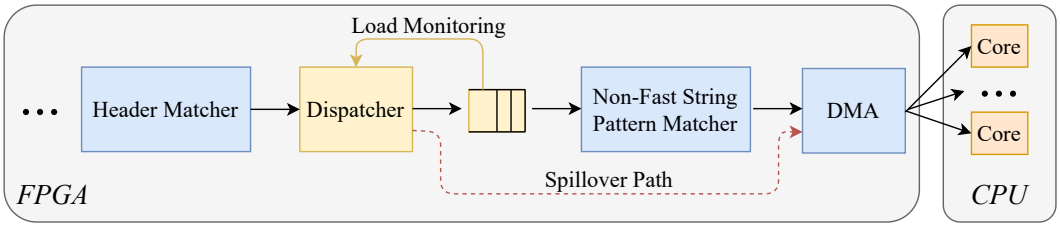


Figure 8.2: NFPSM CPU spillover design

will only send data to NFSPM; the cores that handle the spillover traffic can now be released. We highlight two important details. First, when overflow occurs, both NFPSM and the CPU are working; we are not completely bypassing NFPSM. Second, the spillover traffic that is not processed by NFPSM will not break the correctness of the system, as the CPU will perform a full evaluation of each packet-rule pair. The existence of NFPSM is a performance/resource optimization. By temporarily using a few more CPU cores, our system becomes more resilient to the dynamic workload.

8.2.2 Potential Spillover Use Cases

The NFPSM CPU spillover is just one use case of the dynamic spillover mechanism we consider in the current implementation of Pigasus. Nevertheless, this idea can be applied to any subsystems that are susceptible to high (and unpredictable) variance in traffic. For example, we can also apply this idea to the Header Match stage or the OOO engine in the Reassembly stage. Furthermore, instead of using the CPU for dynamically handling the spillover traffic, we can use Partial Reconfiguration (PR) or even route the traffic to a separate backup FPGA. Below, we discuss how PR and multi-FPGA can potentially help.

Partial reconfiguration: Partial Reconfiguration allows users to partially change

the design at runtime by replacing a particular subcomponent of the running FPGA design with a different, pre-compiled subcomponent without interfering with the rest of the design. With this capability, we can dynamically instantiate a secondary NFPSM PR instance to process the spillover traffic as illustrated in Figure 8.3. This PR instance does not necessarily *have to* have the same configuration as the primary NFPSM. Depending on the intensity of the burst and its duration, one can instantiate different sizes of NFPSM PR to best suit the needs of the spillover traffic.

The challenge is that today, PR takes place at 10-100 milliseconds, a few orders of magnitude higher than the target response time of 100 microseconds – the maximum time our packet buffer can sustain without packet drops at 100Gbps. This suggests that we should carefully manage how PR instances (either swaps in or out) are scheduled. For instance, we should prefetch the PR instance as the load ramps up, using CPU-based compute to handle the spillover traffic in the interim (*i.e.*, while PR takes place).

One caveat of the PR-based solution is that it assumes the space is available when needed. In the case of a single application running aboard the FPGA, the unused portion of the FPGA remains idle. Therefore, the PR solution makes more sense in a multi-tenant environment where the FPGA is multiplexed spatially and temporally among many users. If Pigasus needs a secondary NFPSM, then a low priority task of a different user can be opted out temporarily in order to swap in the required NFPSM filter.

Multi-FPGA: Another approach for implementing dynamic spillover is to use multiple FPGAs. In this model, a backup FPGA will be repurposed at runtime to handle the spillover traffic from the primary FPGA. The traffic can be routed through the inter-FPGA communication channel, such as Ethernet and PCIe. This

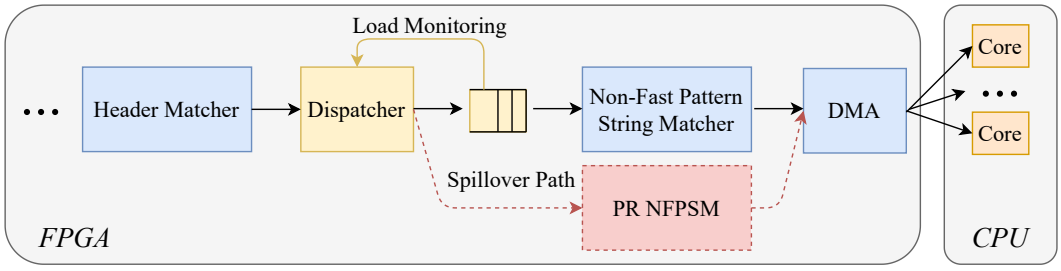


Figure 8.3: NFPSM Partial Reconfiguration spillover design

approach uses the same idea and encounters the same challenges as the PR approach but at a different scale.

8.2.3 Easy Development

Pigasus 2.0's disaggregated service-oriented design methodology enables users to easily explore the many design alternatives for dynamic spillover presented above.

In Pigasus 2.0, the traffic dispatching, traffic merging, and inter-FPGA communication that are frequently used by the dynamic spillover approach are all encapsulated as infrastructural streaming services. Users only need to specify the composition of these infrastructural services with processing services (*e.g.*, NFSPM) in Python to quickly generate various target designs. In our own experience, we were able to generate the RTL implementations of the above PR and multi-FPGA variants within *minutes*.

Currently, users are still responsible for tuning the spillover control policies for the particular design, *e.g.*, deciding when and how to spill over, manually adding PR support for the secondary NFPSM instance, *etc.*

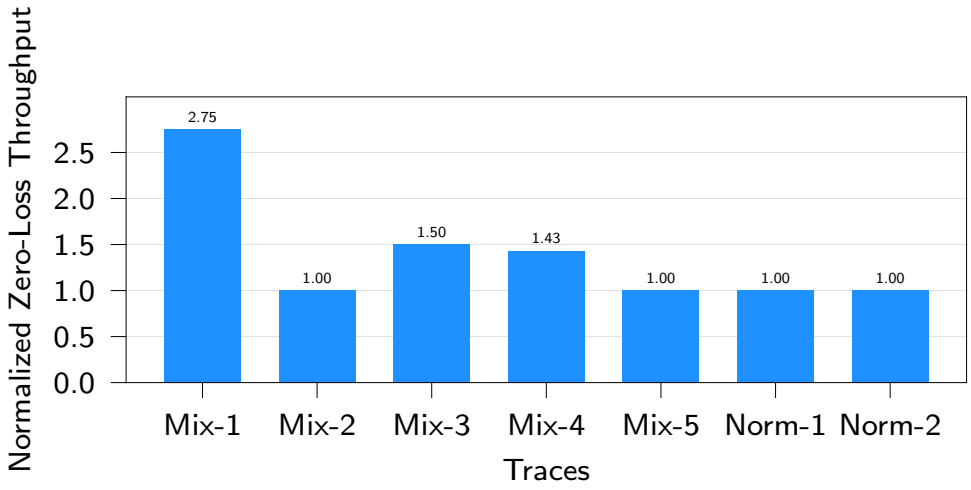


Figure 8.4: Improvement in zero-loss throughput for the spillover design relative to baseline (*i.e.*, no spillover handling)

8.3 Evaluation

Setup: In this section, we evaluate the effectiveness of the spillover mechanism of our NFPSM CPU Spillover implementation. We measure the zero-loss throughput of two designs: one with spillover, and one without. We increase the ingress packet rate until we observe a packet drop. The rest of the setup is identical to Section 3.3.1.

Performance comparison of with and without spillover: Figure 8.4 illustrates the zero-loss throughput comparison of designs with and without spillover handling.

As we can see, for some of the traces (*i.e.*, mix-2, mix-5, norm-1, and norm-2), adding spillover does not improve the performance. That is because for these traces, the NFPSM is not oversubscribed. However, for the bursty traces (*i.e.*, mix-1, mix-3, and mix-4), adding spillover can improve the performance by 1.43-2.75×. This behavior is consistent with Figure 8.1, where in mix-1, mix-3, and mix-4, the

NFPSM is insufficient to handle the bursty traffic, making the FPGA datapath the performance bottleneck. After we enable the spillover path, the spillover traffic can be dynamically processed by the CPU, enabling the design to operate at line-rate.

Chapter 9

Conclusion

Traditional FPGA acceleration developments favor static, fixed-performance designs. This straightforward strategy works well for input-independent streaming applications, such as machine learning inference [6, 7], signal processing [4, 5], and image/video processing [8, 71, 9], where the performance does not depend on the *contents* of the input. However, such designs do not fit input-dependent streaming applications such as IDS/IPS, where different inputs (*i.e.*, packets) trigger different operations leading to variable performance. When there is a large gap between the worst case and the common case, existing fixed-performance FPGA designs that over-provision the resources for the worst case are wasteful, inefficient, and, in many cases, prohibitively expensive. This thesis investigates the question of how to efficiently handle input-dependent streaming on FPGAs in the context of IDS/IPS. This work demonstrates that we should *tune for the common case but adapt the architecture as the input changes.*

In the first part of this thesis, we focused on the common-case tuning. The key idea is to judiciously balance the resource allocation – allocating more resources

to improve common-case performance and using just enough resources for the uncommon case. In particular, we apply this idea to three places in Pigasus IDS/IPS. First, at a system level, we propose an 'FPGA-first' architecture that uses FPGA as the primary processing unit to handle the common case (innocent packets) entirely on a highly-parallel FPGA datapath, while leaving uncommon case (suspicious packets) to the CPU, which works as a complexity offloader to FPGA. In this way, we save significant resources that would otherwise be used to implement expensive (yet relatively infrequently used) regular expression matching on the FPGA. However, this alone is insufficient because 'FPGA-first' requires mapping both Reassembly and MSPM on FPGA while supporting 100Gbps, 100K flows, and 10K rules, which requires excessive BRAM resources using existing designs. Therefore, for the Reassembler, we propose a fast-slow path design that allows the common case (in-order packets) to fly through with high and deterministic performance, and offload the uncommon case (out-of-order packets) to a slower but memory-efficient engine. Finally, for the MSPM, we introduce a hierarchical filter that replicates compact filters in front to process the common case (all packets with all rules), enabling us to use fewer replicas of the more resource-intensive, backend filters to handle the uncommon case (suspicious packets with fewer partially matched rules). Pigasus, the end-to-end FPGA-based IDS/IPS we developed based on the above ideas, can achieve 100Gbps using 1 FPGA and on average 5 CPU cores. This is 100× faster than a CPU-only baseline and 50× faster than existing FPGA designs.

However, the consequence of tuning for common-case performance is that we may overfit to the empirical traces used to guide the design. If the traffic profile of a real deployment does not match the empirical traces, variability in the traffic stream will inevitably result in either performance degradation (due to under-provisioning)

or resource wastage (due to over-provisioning). To address this brittleness, in the second part of this thesis, we proposed two ideas, a disaggregated service-oriented streaming architecture and a dynamic spillover mechanism, to adapt to changing inputs at both compile time and runtime. In the context of our disaggregated architecture, we argue that to handle input-dependent behaviors efficiently, instead of relying on a 'one-size-fits-all' design, developers should create a *space* of designs where the users can select the most efficient one at compile time. To achieve this goal, the design should be disaggregated for efficient scaling, parameterized for easy re-tuning, and connected to a common abstraction for better portability and design productivity. However, this compile-time re-targeting is insufficient to handle the dynamism of the traffic. To address this final shortcoming, we propose a dynamic spillover mechanism that temporarily brings up backup streaming services at runtime to migrate the performance bottleneck.

Overall, through the development of Pigasus IDS/IPS, we demonstrate that we should tune for the common case but adapt the architecture as input changes. Looking forward, we believe our insights and successes can more broadly inform other input-dependent streaming applications beyond IDS/IPS.

9.0.1 Limitations and Future Directions

While Pigasus has demonstrated the effectiveness of our approach of handling input-dependent streaming on FPGAs, there are still some important questions to be answered. Here, we discuss the key limitations of Pigasus and the new research opportunities it enables.

Application protocol parsing: Application protocol parsing is responsible for taking a continuous bytestream and truncating it into application-specific messages

(*e.g.*, HTTP, DNS). Supporting application-level protocol parsing is critical for real-world IDS/IPS deployment, since about 98.2% of the rules in the Snort ruleset [17] are designed for checking application-level messages instead of individual packets.

Parsing application-level messages at line-rate efficiently is challenging. Different from transport-level parsing, which only needs to analyze a single packet at a time, application-level parsing must parse messages from a continuous bytestream, making it possible for a message to span across multiple packets. In order to keep up with line-rate, the protocol parser must sit on the FPGA datapath – routing data back and forth between the CPU and the FPGA for every packet would be impractical.

To implement the protocol parser on the FPGA, one could develop and optimize a protocol parser specifically for each application. However, this is both time-consuming and space-inefficient. Our preliminary analysis of different application protocols suggests that an overlay approach is promising. In particular, we find that these protocols share similar parsing mechanisms. For example, a parser might first search for some known keywords and then extract the message length to infer the message boundary. Because of the commonality, it is possible to use a handful of tailored instructions to perform certain actions, such as searching keywords and counting scanned bytes. Therefore, developing a framework that can execute these instructions at line-rate would be sufficient to parse messages from many application protocols.

Besides IDS/IPS, an application-level parser is also useful for other scenarios. For example, applications that currently rely on TCP stack offloading could be further optimized by also pushing some of their message parsing to an FPGA-enhanced NIC.

Quality of service: Since Pigasus serves on the front-lines of network defenses, it is a prime target for attackers seeking to disrupt network service via Denial-of-Service (DoS) attacks. Broadly speaking, DoS attacks seek to consume as much of a target system's resources (*e.g.*, I/O, memory, compute) as possible, thereby inhibiting its ability to serve legitimate user traffic. In order to guarantee high-performance service *at all times*, it is important to design not just for the common (or average) case, but also for the *worst* case (*i.e.*, in an adversarial setting).

In most rule-based IDS/IPS implementations, the TCP Reassembly engine tends to expose the most significant attack surface [54]; this is also true for Pigasus. As described earlier, Pigasus performs bytestream reconstruction for out-of-order TCP flows using linked lists. While this achieves high memory density (limited FPGA memory being a key design consideration for Pigasus), it also enables an adversary to induce large amounts of wasteful work in the system by creating heavily out-of-order flows, starving innocent user traffic in the process. While Pigasus' 'fast-slow' path design helps protect innocent, in-order traffic from these malicious flows, innocent, out-of-order flows are prone to significant throughput degradation.

Prior works attempt to address this problem in one of two ways: (1) using *ad hoc* heuristics, *e.g.* restricting service to flows with *at most one* concurrent out-of-order packet train [54] or *no more than k* (a constant number of) out-of-order segments [29], or (2) explicitly identifying and dropping malicious traffic. Unfortunately, neither of these strategies is sufficiently general and/or foolproof. For instance, using the heuristics described above greatly inhibits the system's ability to serve moderately out-of-order flows, *even when the system is underloaded*. Also, as exemplified by [53, 72], anomaly detection engines used to classify traffic as innocent/malicious are often easily thwarted by a knowledgeable adversary.

While the 'right' mitigation strategy remains an open question, ongoing efforts suggest that adversarial scheduling theory yields a promising new direction. In particular, by prefacing the system with a carefully-designed scheduler (which explicitly considers an adversary's optimal decisions), we may be able to impose theoretical bounds on the worst-case 'harm' an adversary can induce. Furthermore, another potential complementary approach is to utilize the PR technique to dynamically resolve adversary traffic.

Multi-FPGA acceleration: With the observation that network line-rates are hitting 400Gbps, a natural question to ask is how to scale up IDS/IPS performance beyond 100Gbps. While per-FPGA resource counts are steadily increasing, it is not at a rate that is commensurate with networked application demands. Therefore, using multiple FPGAs may be the only way that we can continue to scale accelerators.

However, today, mapping resource-intensive applications to multiple FPGAs is cumbersome. Developers have to manually partition the design and map components to the right FPGA based on availability of resources, each component's resource requirements, inter-component communication requirements, and inter-FPGA communication capacity. Besides the manual partition, developers often need to implement purpose-built inter-FPGA communication infrastructure. This results in poor productivity when mapping applications onto multiple FPGAs.

One possible solution is to take the application RTL code and automatically generate a feasible partition. This partition must consider the constraints of the multi-FPGA platform as well as the amount of resources needed to meet the performance target. The multi-FPGA communication infrastructure should also be automatically generated for developers. In the end, developers should ideally be

able to map their applications onto multiple FPGAs as easily as mapping it onto a 'single and big' FPGA.

Interestingly, the need for multi-FPGA acceleration is not unique to an IDS/IPS like Pigasus. Any application that cannot fit within the footprint of a single FPGA, *e.g.*, persistent ML inference [6] and large-scale ML training, can benefit from the same solution.

Appendix A

Pigasus Artifact Appendix

A.1 Abstract

Pigasus has a hardware component that runs on an FPGA and a software component which is adapted from Snort 3. The current version requires a host with a multi-core CPU and an Intel Stratix 10 MX FPGA (with 100Gb Ethernet) [41]. Pigasus' artifacts are open-source and publicly available.

We provide detailed instructions to reproduce Figure 3.2. This figure supports our main claim that Pigasus requires two orders of magnitude fewer cores than the state-of-the-art Snort 3. In addition to the steps in this appendix and on the repository README, we also provide video archives that reproduce Figure 3.2 for both the Snort 3 Baseline[1] and the Pigasus[2] experiments.

A.2 Artifact Checklist

- **Algorithm:** Pigasus Multi-String Pattern Matcher.

[1] https://figshare.com/articles/media/snort_baseline_mp4/12922160
[2] https://figshare.com/articles/media/pigasus/12922178

- **Program:** Snort 3 [73] for baseline experiments; DPDK pktgen [44] and Moongen [45] to generate packets.
- **Compilation:** Intel Quartus Prime [74].
- **Dataset:** Stratosphere Laboratory Datasets [46].
- **Run-time environment:** System running Linux with Snort 3 [73] software dependencies installed. Quartus 19.3 with Intel Stratix 10 device support is required to load the bitstream to the FPGA.
- **Hardware:** Two servers, one with an Intel Stratix 10 MX FPGA [41] and another with a DPDK-compatible 100Gb NIC. Power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [48]) or an external electricity usage monitor.
- **Execution:** Disable power optimizations in the BIOS, isolate cores from the Linux scheduler, and pin processes to cores.
- **Experiments:** Experiments are run manually with Pigasus on one machine and a packet generator on another.
- **Public link:** https://github.com/cmu-snap/pigasus
- **Code licenses:** 'BSD 3-Clause Clear License' for the hardware component and 'GNU General Public License v2.0' for the software component. Check the repository for details.

A.3 Description

How to access

To access the artifact, clone the repository from GitHub:

```
$ git clone https://github.com/cmu-snap/pigasus.git
```

This repository also includes a README with the most up-to-date instructions on how to install and extend Pigasus.

Hardware dependencies

Pigasus requires a host with an Intel Stratix 10 MX FPGA [41]. This host should have PCIe Gen3 or greater and a slot with 16 lanes for the FPGA. Experiments require an extra host equipped with a DPDK-compatible 100Gb NIC to be used as a packet generator. For the experiments, the two hosts are connected back-to-back. The power-measurement experiments require either a CPU with a power measurement interface (*e.g.*, RAPL [48]) or the use of an external electricity usage monitor.

Software dependencies

Pigasus' software component is adapted from Snort 3 [73] and inherits the same software dependencies. Appendix A.4 provides instructions on how to install those. The provided implementation works on Linux only and was tested on Ubuntu 16.04 and 18.04. Experiments require the installation of vanilla Snort 3, for comparison, as well as DPDK pktgen and Moongen in the packet generator host. To be able to load the bitstream on the FPGA, an installation of Quartus 19.3 as well as the Stratix 10 device support are required.[3]

Data sets

To obtain the Stratosphere traces, go to https://www.stratosphereips.org/datasets-overview.

[3]Both can be obtained at: https://fpgasoftware.intel.com/19.3/.

A.4 Installation

These instructions assume that you already have the bitstream to be loaded on the FPGA. For instructions on how to synthesize the design, refer to the repository README.

Software configuration

In a system running a fresh install of Ubuntu 18.04, with the Pigasus repository cloned to the home directory, start by setting the required environment variables and useful aliases by adding the following to your `.bashrc` or equivalent:

```
export pigasus_rep_dir=$HOME/pigasus
export pigasus_inst=$HOME/pigasus_install
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
export LUA_PATH="$pigasus_inst/include/snort/lua/?.lua;;"

alias pigasus="taskset --cpu-list 0 $pigasus_inst/bin/snort"
alias sudo='sudo '
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

Then install the dependencies using the provided script:

```
$ cd $pigasus_rep_dir
$ ./install_deps.sh
```

Once the dependencies are installed, build Pigasus as follows:

```
$ cd $pigasus_rep_dir/software
$ ./configure_cmake.sh --prefix=$pigasus_inst --enable-pigasus --enable
    -tsc-clock --builddir=build_pigasus
```

```
$ cd build_pigasus
$ make -j $(nproc) install
```

Hardware configuration

To load the bitstream, make sure the Quartus tools are in your path by setting the following environment variables in your `.bashrc` or equivalent:

```
# quartus_dir should point to the Quartus installation dir.
export quartus_dir=
export INTELFPGAOCLSDKROOT="$quartus_dir/19.3/hld"
export QUARTUS_ROOTDIR="$quartus_dir/19.3/quartus"
export QSYS_ROOTDIR="$quartus_dir/19.3/qsys/bin"
export IP_ROOTDIR="$quartus_dir/19.3/ip/"
export PATH=$quartus_dir/19.3/quartus/bin:$PATH
export PATH=$quartus_dir/19.3/modelsim_ase/linuxaloem:$PATH
export PATH=$quartus_dir/19.3/quartus/sopc_builder/bin:$PATH
```

Make sure you apply these changes:

```
$ source ~/.bashrc
```

A.5 Evaluation and Expected Result

In what follows, we describe how to run the experiments to reproduce Pigasus results from Figure 3.2. Before every experiment, we reload the bitstream on the FPGA and reboot the server. This ensures that we always start from the same FPGA state:

```
$ cd $pigasus_rep_dir/pigasus/hardware/hw_test/
$ ./load_bitstream.sh
```

```
$ sudo reboot
```

Once the machine is back, to run the software component, first insert the kernel module:

```
$ cd $pigasus_rep_dir/software/src/pigasus/pcie/kernel/linux
$ sudo ./install
```

Then, run Pigasus, using the following command:

```
$ cd $pigasus_rep_dir/software/lua
$ sudo pigasus -c snort.lua --patterns ./rule_list
```

The `snort.lua` uses the same syntax as in Snort 3; you should modify it to include the Snort Registered Rule Set [17]. In our experiments, we modified the rules to remove some features currently not supported by Pigasus, including `services`, `file_data`, and `nocase`. We also use the same modified rules in the baseline experiment.

When Pigasus finishes the startup process, it will stop printing logs to the screen. Once this happens, you can invoke the FPGA JTAG console to configure the FPGA internal state. To do so, open another terminal and enter:

```
$ cd $pigasus_rep_dir/hardware/hw_test/
$ ./run_console
% source path.tcl
```

If the last command produces an error, exit the JTAG console with `Ctrl+C` and rerun the last two commands. Once the last command runs successfully, type the following commands to configure the buffer size, set the number of cores, and check the FPGA internal state:

```
% set_buf_size 262143
```

```
% set_core_num 1
% get_results
```

This last command should return all zeros, as no packets have been sent yet.

Now that Pigasus is running and properly configured, we can start the packet generator on another machine. Here, we assume that DPDK pktgen is properly configured on the other machine and has been started.

You can specify the rate to send packets, where 100 means 100% line-rate. To ensure that DPDK pktgen will only send the trace once, specify the number of packets to match the trace size. The example pcap we are using is the `norm-2.pcap`, which has 456,709 packets. After setting these parameters, you can start sending packets.

```
Pktgen:/> set 0 count 456709
Pktgen:/> set 0 rate 100
Pktgen:/> str
```

Once the packet generator finishes sending packets, go back to the JTAG console on the other host and type the following:

```
% get_results
```

This should return 456,709 received packets and 456,709 processing packets. This means that Pigasus processed all of the packets sent at max rate, without loss.

Now stop Pigasus by going back to the first terminal and typing `Ctrl+C`. It will print `rx_pkt`, which should match the `dma_pkt` reported by the FPGA in the second terminal. This means that all packets sent from the FPGA to the CPU for full evaluation were processed.

A.6 Experiment Customization

Experiments may be customized to use different rulesets, different packet traces, and even different designs. To allow easy design exploration for single-FPGA and two-FPGA (connected through 100Gbps Ethernet) scenarios, we provide two Python files, which can generate the new global parameter file `struct_s.sv` and the new top-level file `top.sv` (`top_0.sv` and `top_1.sv` for the two-FPGA scenario) in the `build` directory after running the following commands:

```
$ cd $pigasus_rep_dir/hardware/fluid/
$ python3 ./pigasus/pigasus.py
$ python3 ./pigasus/pigasus_multi.py
```

To customize parameters, edit the PARAMETER CONFIGURATION section in `pigasus.py` or `pigasus_multi.py`. To customize connections, edit the CONNECTIONS section in `pigasus.py` or `pigasus_multi.py`. We provide a commented guide in both Python files to explore different design options. For instance, in `pigasus.py`, you can remove the Header Match stage, and in `pigasus_multi.py`, you can select among the 4 premade cutting points.

To view the visualized connection graph of the top level, uncomment the visualizer pass line in the pass manager at the end of the Python file. To generate the visualizations, you need to have `graphviz` and `PyGraphviz` installed. The generated figure `services.gv.png` is in the following directory:

```
$ cd $pigasus_rep_dir/hardware/fluid/
```

A.7 Artifact Evaluation Methodology

Submission, reviewing, and badging methodology: https://www.usenix.org/conference/osdi20/call-for-artifacts

Bibliography

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, (Renton, WA), pp. 51–66, USENIX Association, Apr. 2018. (document), 1.1, 2.2, 4.1

[2] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern CPUs," in *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, (Boston, MA), pp. 631–648, USENIX Association, Feb. 2019. (document), 1.1, 1.1, 1.2, 2.1, 2.4.3, 4.2

[3] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks.," in *Lisa*, vol. 99, pp. 229–238, 1999. (document), 1, 2.2, 2.1

[4] C. Ribeiro and A. Gameiro, "A software-defined radio fpga implementation of

ofdm-based phy transceiver for 5g," *Analog Integr. Circuits Signal Process.*, vol. 91, p. 343–351, May 2017. 1, 9

[5] X. Cai, M. Zhou, and X. Huang, "Model-based design for software defined radio on an FPGA," *IEEE Access*, vol. 5, pp. 8276–8283, 2017. 1, 9

[6] M. Hall and V. Betz, "Hpipe: Heterogeneous layer-pipelined and sparse-aware cnn inference for fpgas," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 320, Association for Computing Machinery, 2020. 1, 9, 9.0.1

[7] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), p. 161–170, Association for Computing Machinery, 2015. 1, 9

[8] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-based real-time pedestrian detection on high-resolution images," in *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 629–635, 2013. 1, 9

[9] J. Zhang, J. Li, and S. Khoram, "Efficient large-scale approximate nearest neighbor search on OpenCL FPGA," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4924–4932, 2018. 1, 9

[10] "Zeek - network security monitor." https://www.zeek.org. 1, 2

[11] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999. 1, 2

[12] "Suricata." https://suricata.io/. 1

[13] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012. 1

[14] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. of NSDI 2012*, NSDI'12, (Berkeley, CA, USA), pp. 24–24, USENIX Association, 2012. 1

[15] M. Branscombe, "The year of 100GbE in data center networks." https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks, Aug. 2018. 1.1, 2.1

[16] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, (New York, NY, USA), p. 267–280, Association for Computing Machinery, 2010. 1.1, 2.1

[17] "Snort ruleset." https://www.snort.org. 1.1, 2.1, 3.3.1, 9.0.1, A.5

[18] Z. K. Baker and V. K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," in *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, ANCS '05, (New York, NY, USA), pp. 193–202, Association for Computing Machinery, 2005. 1.1, 2.4.1, 4.1, 4.2

[19] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar, "Deep packet inspection in FPGAs via

approximate nondeterministic automata," in *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '19, pp. 109–117, 2019. 1.1, 1.2, 2.4.1, 2.4.3, 4.1, 4.2, 6.2, 1

[20] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, pp. 249–257, 2004. 1.1, 2.4.1, 4.1, 4.2

[21] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," *IEEE Micro*, vol. 24, p. 52–61, Jan. 2004. 1.1, 2.4.1, 4.1, 4.2

[22] H. Song, T. Sproull, M. Attig, and J. Lockwood, "Snort offloader: a reconfigurable hardware NIDS filter," in *International Conference on Field Programmable Logic and Applications*, FPL '05, pp. 493–498, 2005. 1.1, 2.4.1, 4.1, 4.2

[23] P. Orosz, T. Tóthfalusi, and P. Varga, "FPGA-assisted DPI systems: 100 Gbit/s and beyond," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 2015–2040, 2019. 1.1, 2.4.1, 4.1, 4.2

[24] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *32nd International Symposium on Computer Architecture*, ISCA '05, pp. 112–122, June 2005. 1.1, 2.4.1, 4.1, 4.2

[25] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 227–238, 2001. 1.1, 2.4.1, 4.1, 4.2

[26] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare:

Hardware accelerator for regular expressions," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016. 1.1, 2.4.1, 4.1, 4.2

[27] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "Reapr: Reconfigurable engine for automata processing," pp. 1–8, 09 2017. 1.1, 2.4.1, 4.1, 4.2

[28] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 36–43, 2015. 1.1, 1.2, 2.4.1, 2.4.2, 4.1, 4.2, 5.2

[29] R. Yuan, Y. Weibing, C. Mingyu, Z. Xiaofang, and F. Jianping, "Robust tcp reassembly with a hardware-based solution for backbone traffic," in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, pp. 439–447, 2010. 1.1, 1.2, 2.4.1, 2.4.2, 4.1, 4.2, 5.2, 9.0.1

[30] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 171–186, USENIX Association, Apr. 2018. 1.2, 2.4.1, 4.1, 4.1, 4.2

[31] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, pp. 333–340, June 1975. 2.1, 4.2

[32] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on*

Networked Systems Design and Implementation, NSDI '16, (Santa Clara, CA), pp. 551–565, USENIX Association, Mar. 2016. 2.1

[33] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable router architecture for experimental research," in *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, (New York, NY, USA), pp. 1–7, Association for Computing Machinery, 2008. 2.2

[34] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: energy-efficient microservices on SmartNIC-accelerated servers," in *2019 USENIX Annual Technical Conference USENIX ATC 19*, pp. 363–378, 2019. 2.2

[35] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé, "Fast string searching on PISA," in *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, (New York, NY, USA), pp. 21–28, Association for Computing Machinery, 2019. 2.2

[36] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), pp. 1–14, Association for Computing Machinery, 2016. 2.2

[37] BERTEN, "GPU vs FPGA performance comparison." http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf. 2.2

[38] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park,

"Kargus: A highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 317–328, Association for Computing Machinery, 2012. 2.2

[39] H. Liu, S. Pai, and A. Jog, "Why GPUs are slow at executing NFAs and how to make them faster," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), pp. 251–265, Association for Computing Machinery, 2020. 2.2

[40] A. Subramaniyan and R. Das, "Parallel automata processor," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 600–612, Association for Computing Machinery, 2017. 2.2

[41] "Intel Stratix 10 MX." https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html. 2.2, 3.3.1, 3.3.2, 3.3.2, A.1, A.2, A.3

[42] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pp. 138–147, IEEE, 2020. 2.4.1, 4.2

[43] M. Nguyen, *Dynamically Managing FPGAs for Efficient Computing*. PhD thesis, Dec 2020. 2.4.4, 7.1

[44] "DPDK-pktgen." https://github.com/Pktgen/Pktgen-DPDK. 3.3.1, A.2

[45] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, pp. 275–287, 2015. 3.3.1, A.2

[46] Stratosphere, "Stratosphere laboratory datasets," 2015. Retrieved March 13, 2020, from https://www.stratosphereips.org/datasets-overview. 3.3.1, 7, A.2

[47] S. Garcia, "Modelling the network behaviour of malware to block malicious patterns. the stratosphere project: a behavioural IPS," *Virus Bulletin*, pp. 1–8, 2015. 3.3.1

[48] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012. 3.3.2, A.2, A.3

[49] "PN1500 Watt meter." https://poniie.com/products/17. 3.3.2

[50] "Microsoft Azure: Total cost of ownership (TCO) calculator." https://azure.microsoft.com/en-us/pricing/tco/calculator/. Accessed: 2020-10-01. 3.3.2

[51] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1083–1100, USENIX Association, Nov. 2020. 3.3.2, 5

[52] CAIDA. https://www.caida.org/research/traffic-analysis/AIX/plen_hist/. 3.3.3, 5.3.2

[53] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *10th USENIX Security Symposium (USENIX Security 01)*, (Washington, D.C.), USENIX Association, Aug. 2001. 5.1, 9.0.1

[54] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries," in *14th USENIX Security Symposium (USENIX Security 05)*, (Baltimore, MD), USENIX Association, July 2005. 5.2, 9.0.1

[55] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th {USENIX} Symposium on Networked Systems Design and Implementation {NSDI} 19*, pp. 531–548, 2019. 5.3.3

[56] Y. Arbitman, M. Naor, and G. Segev, "De-amortized cuckoo hashing: Provable worst-case performance and experimental results," in *International Colloquium on Automata, Languages, and Programming*, pp. 107–118, Springer, 2009. 5.3.3

[57] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2010. 5.3.3

[58] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 27, p. 67–82, July 1997. 5.4

[59] T. T. Hieu and N. T. Tran, "A memory efficient FPGA-based pattern matching engine for stateful NIDS," in *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 252–257, 2013. 6.2

[60] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 112–122, 2005. 6.2

[61] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, (New York, NY, USA), p. 50–59, Association for Computing Machinery, 2008. 1

[62] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35, p. 74–82, Oct. 1992. 6.2

[63] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, p. 422–426, July 1970. 3

[64] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: An efficient data structure for static support lookup tables," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, (USA), pp. 30–39, Society for Industrial and Applied Mathematics, 2004. 3

[65] M. K. Papamichael, *Pandora: Facilitating IP Development for Hardware Specialization*. PhD thesis, Aug 2015. 7.2

[66] J. Melber and J. C. Hoe, "A service-oriented memory architecture for FPGA computing," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 91–97, 2020. 7.2, 7.3.3

[67] "Avalon interface specifications." https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf. 7.3.1

[68] "AXI reference guide." https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. 7.3.1

[69] "Crossroads 3D-FPGA." http://www.crossroadsfpga.org/. 7.3.3

[70] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016. 7.3.3

[71] S. Wang, C. Zhang, Y. Shu, and Y. Liu, "Live video analytics with FPGA-based smart cameras," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo'19, (New York, NY, USA), p. 9–14, Association for Computing Machinery, 2019. 9

[72] P. Fogla and W. Lee, "Evading network anomaly detection systems: Formal reasoning and practical techniques," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (New York, NY, USA), p. 59–68, Association for Computing Machinery, 2006. 9.0.1

[73] "Snort 3." https://www.snort.org/snort3. A.2, A.3

[74] Intel, "FPGA design software – Intel Quartus Prime." https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. Accessed: 2020-10-14. A.2