Shire: Making FPGA-accelerated Middlebox Development More Pleasant

Moein Khazraee[*] *Alex Forencich*
George Papen, Alex C. Snoeren, Aaron Schulman
[*]*Massachusetts Institute of Technology* *UC San Diego*

Abstract

We introduce an approach to designing FPGA-accelerated middleboxes that simplifies development, debugging, and performance tuning by decoupling the tasks of hardware accelerator implementation and software application programming. Shire is a framework that links hardware accelerators to a high-performance packet processing pipeline through a standardized hardware/software interface. This separation of concerns allows hardware developers to focus on optimizing custom accelerators while freeing software programmers to reuse, configure, and debug accelerators in a fashion akin to software libraries. We show the benefits of Shire framework by building a firewall based on a large blacklist and porting the Pigasus IDS pattern-matching accelerator in less than a month. Our experiments demonstrate Shire delivers high performance, serving ∼200 Gbps of traffic while adding only 0.7–7 microseconds of latency.

1 Introduction

FPGAs have become the preferred platform for high-speed packet processing due to their flexibility: middleboxes perform a wide variety of network functions, many of which require hardware acceleration to function at today's line rates. For example, intrusion detection systems (IDS) leverage hardware acceleration for signature matching [17, 31] and SD-WAN middleboxes [15] rely on hardware acceleration to support encrypted tunnels. Critically, FPGAs—as opposed to ASICs—also make it possible for vendors to update their middleboxes after deployment. For example, release notes from Palo Alto Networks indicate they have patched a variety of FPGA bugs in their deployed firewalls [16].

Despite their popularity, there are several challenges to effectively employing FPGAs in network middleboxes. Perhaps the most fundamental is the need to parallelize operations to meet the throughput requirements of line-rate packet processing. FPGA logic operates 4–10× slower than ASICs (e.g, 250 MHz vs. 1–3 GHz). Therefore, to achieve 100-Gbps or greater throughput, middlebox developers are forced to arrange individual accelerators into a carefully crafted parallel packet processing pipeline. This elaborate orchestration also makes maintaining FPGA middleboxes challenging because any changes to accelerator implementation or middlebox logic will likely require a hardware programmer to redesign the entire pipeline to maintain the same performance.

Unfortunately, existing FPGA packet processing frameworks such as ClickNP [11] and HxDP [3] do not address this challenge: they still require developers to manually orchestrate packet processing pipelines—a task typical network software programmers are ill-equipped to handle. Indeed, Microsoft "found the most important element to successfully programming FPGAs has been to have the hardware and software teams work together in one group" [5]. As further evidence of the complexity of this task, the Pigasus 100-Gbps IDS middlebox [31] took two years to develop.

We present Shire, a programming framework that both simplifies the development and improves the performance of middleboxes that employ custom FPGA-based hardware accelerators. Shire abstracts away the packet-processing pipeline so hardware developers can focus on implementing accelerators following a familiar packet-based DMA interface. Software programmers can then integrate these accelerators into packet-processing pipelines on RISC-V cores through a C-based API (Section 3). Scheduling and data movement are managed by the Shire framework and implemented in our runtime that meets the physical constraints of large FPGAs (Section 4). Shire leverages the packetized nature of network traffic to achieve line rate. Packets represent natural units of concurrency and Shire allows the runtime to exploit them without requiring developers to explicitly manage it. Moreover, by invoking multiple instances of the same accelerator in a pipelined or concurrent fashion, Shire increases the number of cycles available to a hardware designer.

Shire makes it possible to use existing hardware accelerators to process network traffic at 200 Gbps. We implement our runtime on the Xilinx XCVU9P 200-Gbps FPGA and demonstrate that we can instantiate 16 parallel packet processors running at 250 MHz providing a 16× increase in clock cycles available to process each packet (Section 5). For payloads of 128 bytes, Shire can can process—and generate—packets at a line rate of 200 Gbps; 65-byte minimum-sized packets can be forwarded at 173 Gbps. Even at the smallest packet size our framework adds only ∼0.7 μsec (∼7.1 μsec worst case

for jumbo packets) of latency (Section 6). We build a firewall based on a large blacklist and ported the Pigasus IDS's pattern matching accelerator in less than a month (Section 7). Shire is implemented in Verilog with all open-source IP cores; we will release it as open source at the time of publication.

2 Background and motivation

The primary benefit of FPGAs is programmability: middlebox developers can use FPGAs to avoid the development costs associated with creating an ASIC. Historically this programmability came at the cost of performance and capability: FPGAs have slower clocks and smaller logic and memory capacity than ASICs. In this section, we explain why FPGAs have recently become popular for high-performance 100-Gbps+ middlebox implementation and motivate the need for Shire.

2.1 Network-targeted hard logic

Modern FPGA vendors overcame the prior functionality limitations by tailoring FPGA hardware for high-speed packet processing. In particular, FPGAs now contain ASIC-like hardened logic for communication links, e.g., 100-Gbps Ethernet and PCI Express PHY/MAC. They also contain larger (e.g., 8×) memory cells to enable buffering multiple packets, providing needed slack for complex, long-running packet processing (e.g., managing different packet sizes) [28].

These network-specific resources, however, are provided as bare-bones hardware; middlebox developers need to implement their own glue logic from scratch to use them. Shire hides the complexities of using these networking-specific hardware elements by integrating them with the Shire runtime. With Shire, middlebox developers can focus their hardware effort on accelerator design, and integrate their accelerators into the runtime using software.

2.2 Higher density complicates development

Similarly, manufacturers addressed clock-speed limitations by boosting fabric capacity (e.g., logic, memory, and I/O) through cutting-edge techniques that increase transistor density on FPGA chipsets [23]. For instance, Xilinx employs 7-nm CMOS process nodes and multiple (2–3) interconnected FPGA dies to multiply the logic available in a chipset [21]. Middlebox developers can use these additional resources to increase throughput by parallelizing their implementations.

Unfortunately, scaling up FPGA resources introduces an additional challenge for developers: they need to provide hints to the heuristic-based FPGA design tools to help them find a feasible physical layout for their implementation in the FPGA's fabric. Xilinx recommends that developers organize their designs in an hierarchical fashion [29] and try iteratively compiling (i.e., synthesizing) different designs and layouts until they find one where the heuristic-based approach can satisfy the timing and placement constraints of the FPGA's fabric. For large designs typical of middleboxes, each of these compile runs takes multiple hours; therefore development iterations to find a feasible design and layout can take days. These layout decisions are especially difficult when the developer needs to expand their logic to cross die boundaries as die interconnects only exist on certain locations.

Developers are also attracted by the promise of partial reconfiguration—recently termed Dynamic Function Exchange (DFX). This feature enables update of only a portion of the FPGA. Instead of fully pausing the FPGA functionality during load of bit stream, DFX can help developers reach a feasible implementation by initially crafting a portion of the design as a static part, and adding on incrementally with builds for the configurable regions. However, these approaches introduce additional constraints for selection of those configurable regions, the interface to that region, and also ensuring the system remains stable during the partial reconfiguration process.

The Shire framework significantly reduces the need for developers to consider layout of their hardware design, reducing lengthy development cycles. Shire itself only needs to be laid out for a particular FPGA chipset once. Developers design and layout their custom hardware accelerators within the partially reconfigurable regions, dedicated for Shire's packet processors.

2.3 Parallelization is challenging

While the increased real estate of modern FPGA makes parallelization possible, actually realizing a parallel design requires careful orchestration on the part of the middlebox developer. FPGA hardware development languages (i.e., HLS) have the potential to assist, but no existing development tool has the capability to automatically produce performant parallelized implementations that operate at today's highest link rates. Microsoft's HLS middlebox framework, ClickNP [11] relies on proprietary Catapult framework, and only operates within an acceleration unit, called a role. Even still, it requires developers to manually decide how their hardware accelerators are pipelined. More to the point, published reports only document operation at 40 Gbps without partial reconfiguration support, whereas we demonstrate that middleboxes developed with Shire can achieve 200 Gbps.

HLS also has the potential to bridge the gap between software and hardware development, similar to how Shire provides a software-like programming interface for hardware accelerator development. However, HLS is still at its core a tool that generates Verilog, so the developer needs to be fully aware of hardware restrictions and dependencies among different hardware modules when they write their code. Moreover, HLS does not support software-like debug tools, such as break points or memory dumping. FPGA developers need to explicitly integrate debugging hardware into their design if they want visibility into run-time state. Furthermore, HLS

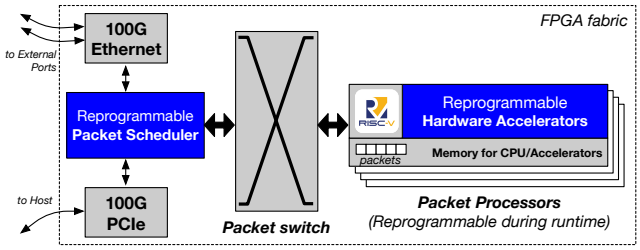


Figure 1: The Shire framework provides a simplified abstraction for FPGA-accelerated middlebox packet processing.

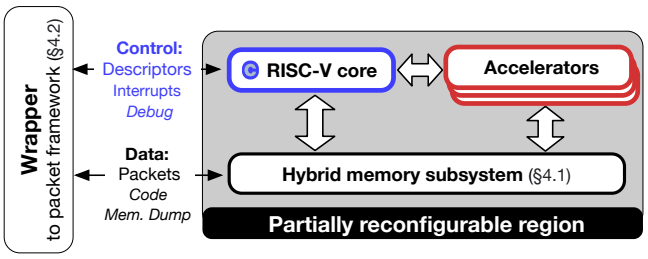


Figure 2: Internal architecture of a Shire packet processor

designs are typically less performant than developing Verilog from scratch [11]. However, developers may find HLS useful in the development of their custom hardware accelerators, and those HLS-generated accelerators can be placed inside Shire's packet processors.

3 Shire abstraction

The centerpiece of Shire is a new hardware abstraction: a reconfigurable RISC-V core connected to a set of custom hardware accelerators, all residing inside a partially reconfigurable hardware blocks. As shown in Figure 1, the Shire runtime orchestrates packet scheduling, allowing the developer to focus on the implementation of packet processing logic. Lightweight functions like parsing and selecting rulesets can still be implemented in software, but run on RISC-V cores instantiated directly on the FPGA where it can invoke custom hardware accelerators when appropriate. If desired, the developer can implement custom packet schedulers in software to assist in steering packets to the appropriate packet processor to meet locality constraints (e.g., to support stateful flow-based functionality).

Shire's C-like programming model benefits all aspects of the development cycle. In production, software running on the RISC-V cores can dynamically configure and invoke hardware accelerators. In addition, software running on the host can dynamically update the accelerators using partial reconfiguration. During development, hardware developers can identify and mitigate issues with accelerators by monitoring the state of accelerators and raising faults that are handled directly on the soft core. Similarly, software programmers can use Shire's simulation framework to test how their code would work with existing accelerators, without needing to lay out a full design and running it on actual traffic.

3.1 Architecture of Packet Processors

Figure 2 shows the internal architecture of a packet processor. Each packet processor contains two primary processing components, a RISC-V core, and a set of accelerators that the core controls. The RISC-V core and the accelerators communicate over two memory-based interfaces: (1) a basic memory-mapped IO for configuring and reading accelerator registers, and (2) a novel hybrid memory subsystem. The hybrid memory subsystem (Section 4.1) provides a unified memory for both the core and accelerators to share *packets* that are currently being processed, as well as the *state* of both the core (e.g,. stack) and the accelerators (e.g., scratchpad), and the *instructions* that the RISC-V core is executing. A "wrapper" module provides the connection between the packet processor and the packet distribution subsystem. These modules enable the Shire framework to schedule and switche packets between packet processors, the FPGA's NIC interfaces, and the host's DRAM (Section 4.2). The wrapper informs the core about arriving packets by giving it descriptors, and the core can send packets back to the packet distribution framework by giving descriptors with the desired destination to the wrapper. The wrapper also provides a control interface from the host to the packet processor, for example, it can read and write the status registers, and interrupt the processor. The entire packet processor resides inside a partially reconfigurable region of FPGA logic, called a PR block, making it possible to replace it with new accelerators, or a reconfigured RISC-V core.

3.2 Goal: Flexibility

Packet processors provide a well-defined interface for software and hardware to interact: the memory interface between the core and the accelerators. As long as hardware and software follow the same interface, either can change, without affecting the operation of the other. Software can tell an accelerator what data to operate on (e.g., packet) by passing a pointer—using memory-mapped I/O—to the data's location in the hybrid memory. Then hardware accelerator uses its port to the hybrid memory to read the data it needs using the address it was given by the software. This effectively brings to FPGAs the modular hardware accelerator interface used in SoCs, with the additional benefit of allowing developers to modify hardware accelerators.

The main challenge with supporting reconfigurability at runtime is transitioning the state from a running packet processor to a new one, without resulting in inconsistent state, or dropping any packets. To do so, the host sends a signal to the packet framework to stop sending packets to that specific packet processor, and interrupts the RISC-V CPU to inform

it to finish processing the current packet and save its state to the host via the packet framework. To reconfigure hardware, the host writes the new bitfile to its PR block in the FPGA. Then, to reprogram software, the host uses the packet framework to to write directly to the RISC-V core's instruction and data memory residing in the hybrid memory subsystem. After reconfiguring the FPGA, the new RISC-V core boots up, restores its state by reading it back from host memory, then the host sends a signal to the packet framework to start sending packets to the core again. If the new hardware was just a bug fix or optimization, and not a functionality change, the RISC-V's execution can start again at the next unprocessed packet, with all of the previous state retained.

3.3 Goal: Debuggability

An additional benefit of our unified memory architecture is the ability to support multiple debugging tools and techniques. For example, at runtime hosts can send an interrupt to tell a packet processor and its accelerators to stop processing, then the host can read the state of both the accelerators and CPU by dumping the entire hybrid memory subsystem. Moreover, our framework provides 64-bit debug registers that allow the host and packet processor to communicate directly. For example, if the hardware hangs (i.e., the packet subsystem), software on the RISC-V can detect the hang, read the accelerator state, and put it on the debug channel.

Shire's architecture also supports a preventive approach to debugging: simulating an entire packet processor's operation, including hardware accelerators, hybrid memory subsystem, and the RISC-V core controlling them. We build a Python-based packet processor test bench framework based on Cocotb. Developers can then link in the hardware accelerators and software they want to test, and run full simulation of the packet processor's operation before deploying on the FPGA. Another benefit of using python is availability of several libraries, such as Scapy [2] that we use to generate test cases. We also provide a python library with the functions available between host and packet processor.

4 Architectural Components

In this section we describe the architecture components that make it possible to provide the Shire abstraction without a significant decrease in link rate or increase in latency over fully customized FPGA implementations. These components are designed in particular to work around the relatively slow clock rates of typical FPGAs (e.g., 250 MHz). The hybrid memory subsystem inside each packet processor makes it possible for a relatively slow RISC-V CPU core to keep up with the high packet rate, and also communicate efficiently with the high-performance hardware accelerator pipelines. Our packet distribution framework facilitates running over a dozen parallel packet processors, delivering a combined throughput of

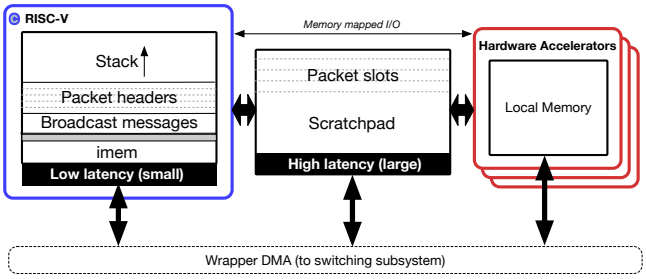


Figure 3: Hybrid memory subsystem in each packet processor

200 Gbps. At the end of this section, we overview the host software interface to these components.

4.1 Memory subsystem

We observe that accelerators and RISC-V cores have different ways of accessing memory. Accelerators usually read the packet payloads in a streaming manner (e.g., word-by-word in sequence). The reason is, often accelerators need to process the entire payload in order, and perform compute-heavy, i.e., relatively time-consuming processing. As a result, for accelerators we can benefit from using the larger higher-latency memories (e.g., Xilinx's "Ultra-RAMs") that can be pipelined to hide the latency, and also keep up with line-rate throughput.

In contrast, the RISC-V cores have more random access; for example parsing a header and deciding on the next field to read based on the output of that first read. This random read pattern is very difficult to support with higher-latency memories. Fortunately, the amount of data in the packet headers that the cores need access to is much smaller. As a result, we can use smaller lower-latency memories (e.g., BRAMs) for the RISC-V cores. This contrast provides an opportunity to design a tailored memory architecture.

Figure 3 shows an overview of our hybrid memory architecture: we split the memory space into three parts. First, there are instruction and data memories of the RISC-V core which are small and can be accessed within a cycle. Then, there is the large packet memory (center), where the packets arrive at from the switching system. This memory is shared between the RISC-V and accelerators, and can be used as a scratch pad. Finally, accelerators can have local memory loaded by the switching subsystem for lookup tables or similar.

The DMA engine inside the wrapper has access to all these memories. This DMA engine is customized to copy an incoming packet to the shared packet memory, and also copy the packet header into the local RISC-V memory. This design enables the RISC-V core to parse the header with low latency, while at the same time several accelerators can access the entire packet from the packet memory in a streaming fashion. It can also be used to initialize the memories from the host—before booting the RISC-V core—to load lookup tables, or

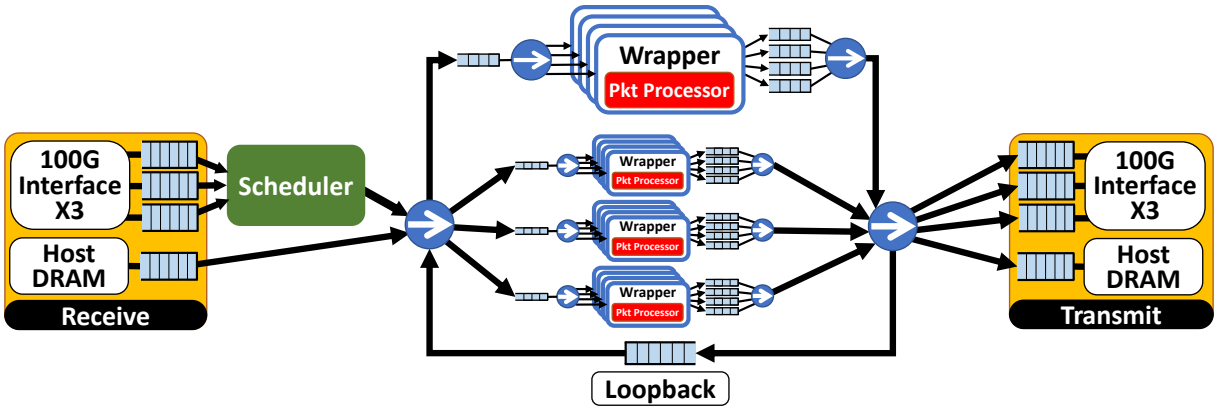


Figure 4: **Switching subsystem overview**

read them back for debugging purposes.

One limitation with FPGA memories is that they only have two ports. To avoid contention, local RISC-V memories have a dedicated port to the core, and the other port is used for the DMA engine to exchange packet-header data and facilitate low latency communication among packet processors (see Section 4.2.2). Since cores sparsely access the packet memory—e.g., for table look-up once per packet or for change a value in packets header—we share that port with the DMA engine and give higher priority to the core. This frees up the other packet memory port for the accelerators to have exclusive access. Finally, both ports of the local accelerator memory are dedicated to the accelerators for faster reads, only during boot or readback where the accelerators are not active, can DMA engine uses one of the ports. For accelerator configuration or result readback, the RISC-V cores uses a separate memory mapped I/O (MMIO) channel to perform writes or reads the accelerators' registers.

4.2 Packet distribution

Shire's packet-distribution subsystem internally moves packets between interfaces and packet processors in an efficient manner. Figure 4 shows an overview of how packets flow from interfaces to packet processors. Incoming packets first arrive at a physical or virtual Ethernet interfaces where the are assigned a destination packet processor by a programmable scheduler. For instance, we can implement a balancing scheduler that assigns a new packet to the least-loaded core. The packet is then sent over a sequence of on-chip switches to the selected packet processor; after processing, another switch carries the packet to an outgoing interface. There are two other possible sources and sinks for packets: (1) Host DRAM: indeed, we reuse the packet switches to move data over PCI-e to and from the host, and (2) loopback: this is used when a packet processor wants to send a full packet to another packet processor, further described in Sec.4.2.2. These interfaces typically carry much less traffic than network-facing interfaces, so they can share the same infrastructure without sacrificing throughput.

As shown in Figure 4, switching is performed in two stages: first among four clusters and then among four packet processors. This is to achieve the required performance with resource utilization in mind. We implement a separate switch for each cluster that has full throughput on one side, and four links running at 1/4th throughput on the other side. By using separate FIFOs for each incoming link inside this switch, we enable non-blocking forwarding: each FIFO provides bit-conversion without blocking the other incoming interfaces. Thus, the only necessary arbitration is when two input interfaces send to the same packet processor. We use round-robin-policy arbitration by default; it can be replaced with more sophisticated policies if necessary.

4.2.1 Packet movement

We leverage the packet-based data flow to separate memory addressing from the switching subsystem. In particular, we abstract away memory addresses and refer to packet data by a descriptor or slot number. Similarly, communication between host DRAM and packet processors is also packetized, using a different slot number, i.e., DRAM tag. The wrapper implements address handling and interfacing with the core, as well as communicating to the scheduler and the host DRAM access manager. In other words, we split the control functionality into a central part—the scheduler—and a distributed part each wrapper, to make it more scalable. Coordination is facilitated by separate control channels for messages among packet processors and packet scheduler, as well as request messages to host DRAM access manager. These control channels are shown in Figure 5. The switches have two stages similar to data-flow switches described in the previous subsection, but there is no bit-width conversion and simple switches are used.

We can now describe how packets actually flow through the system. At boot, software running on RISC-V allocates some slots for packets and notifies the central scheduler about the

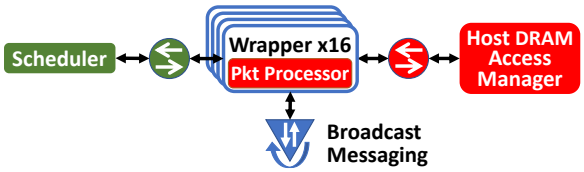


Figure 5: **Control messaging flow**

number of slots and their maximum size. When the scheduler assigns a packet to a packet processor, its wrapper notifies the core of an active a descriptor. When sending out a packet, a RISC-V core has two options: it can ask the wrapper to send it out directly, or it can tell the scheduler which slot is ready to be sent out and scheduler sends the transmit command to the wrapper. In both cases the wrapper notifies the scheduler about slot being freed after it is sent out.

4.2.2 Inter-packet processor messaging subsystem

Sharing state across packet processors can be necessary to implement applications that require stateful packet processing. Unfortunately, due to limited memory and the low clock rate on FPGAs; replicating incoming packets in the internal memory of all cores, or benefiting from shared caches are inefficient and impractical methods to implement.

Our insight is most of the message passing in networked applications can be put into two categories: (1) copying a packet or portion of memory to another core, or (2) a short message to update other cores. In a cached system both of these can be implemented through coherency. However, we can take advantage of our packet subsystem to provide similar functionality. We provide a loopback module that can route a packet from any packet processor to any other packet processor. RISC-V cores can use the provided interface to ask the scheduler for a packet slot of the destination packet processor, and the packet can be transmitted using the same switching subsystem system. Inter-core packet messaging can be used to implement a chain of heterogeneous packet processors with different accelerators and capabilities.

For the second type of messaging, in cached systems a write to a location in shared memory would evict that value from every other core. Later when another core requires that value it gets the updated value from this core. We simplified this to a broadcast system, where portion of memory is semi-coherent: a write to this portion would be propagated to all the other cores, and they would receive the message at the exact same time. This system is shown in Figure 5. This simpler method incurs less contention than a coherent cache. Also a program can select a range in the shared memory address space to get interrupts for arrival of such messages, alongside the address of the incoming data, which helps to separate data and control messages. Shire also provides a FIFO for these addresses not to lose their order and new messages addresses, removing the need to pool the entire shared memory for the new messages.

4.3 Interaction with the framework

Finally we briefly explain different host interfaces available to the packet distribution framework, and how they can be used to achieve our flexibility and debuggability goals.

Interface between the host and switching subsystem. Hosts can read counters for for all the physical and virtual ethernet interfaces, as well as each packet processor. These counters read the number of transferred bytes, frames, drops, or stalled cycles and can shed light to how packets are going through the system, how the scheduler is distributing them or where the bottlenecks are.

Interface between the host and user's scheduler. There is a read/write channel going from host commands to the scheduler, with 30 bits of separate address space for write and read, along side 32 bit write and read values. The user can fully customize this channel to configure and control their scheduler during runtime. For instance, we use this channel to select which cores are used for incoming traffic, which cores are disabled, or to read number of available slots inside the scheduler per packet processor and other status registers. These data helps us to detect freezes or starvation scenarios. Also we can prepare the scheduler for load of a new packet processor by flushing the slots in the scheduler.

Customizing the packet distribution framework. The main flexibility available to developers in the packet distribution framework is to change the scheduler hardware. We provide TCL scripts to make faster incremental builds from the base FPGA image with a new scheduler. We also can customize the number and size of each packet processor; the user just has to select the PR regions in the FPGA and map them in the script to the packet processors.

5 Implementation

Our implementation uses a Xilinx Virtex UltraScale+ FPGA VCU1525 board with an XCVU9P FPGA chip as shown in Figure 6 and Figure 7. There are 16 and 8 packet processor versions respectively, where each packet processor has a unique PR allocation as well as a wrapper next to it for communicating to the rest of the system. There is another larger PR block for the scheduler, labeled Scheduler_block. Note that our current scheduler implementations are basic and require very few resources, and the rest of reserved area is empty for potentially a more sophisticated user scheduler. Reconfiguring scheduler during run time would stop the system, and the main role of partial reconfigurability is to isolate placement and routing. If run time updates more than just a few parameters are required, there could be a replica to switch to during partial reconfiguration.

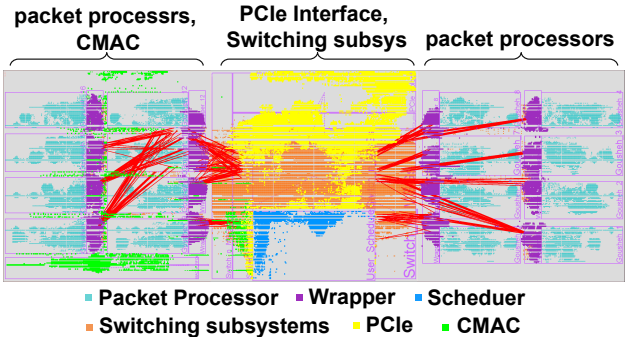


Figure 6: Layout of components of XCVU9P FPGA with 16 packet processors and Firewall accelerator.

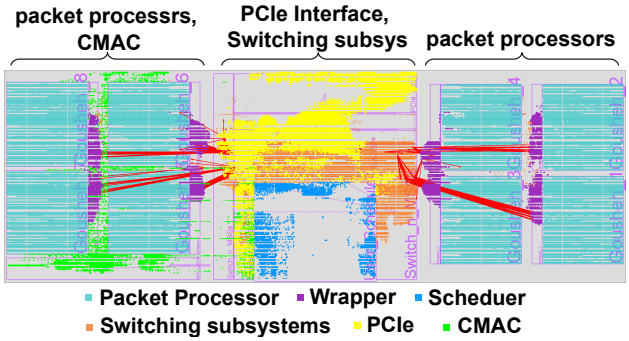


Figure 7: Layout of components of XCVU9P FPGA with 8 packet processors and Pigasus accelerator.

Component	LUTs	Registers	BRAM	URAM	DSP
Single processor	4588 (0.4%)	3755 (0.2%)	24 (1.1%)	32 (3.3%)	0
Remaining (PR)	23251 (2.0%)	52165 (2.2%)	12 (0.56%)	0	168 (2.5%)
Scheduler	9858 (0.8%)	15119 (0.6%)	0	0	0
Remaining	68526 (5.8%)	143281 (6.1%)	144 (6.7%)	48(5.0%)	576 (8.4%)
Single wrapper	2916 (0.2%)	2935 (0.1%)	0	0	0
CMAC	9132 (0.8%)	13804 (0.6%)	0	18 (1.9%)	0
PCIe	42505 (3.6%)	68502 (2.9%)	110 (5.1%)	32(3.3%)	0
Switching	99320 (8.4%)	126530 (5.4%)	48 (2.2%)	64(6.7%)	0
Complete design	280858 (23.8%)	330995 (14.0%)	542 (25.1%)	626 (65.2%)	0
VU9P device	1182240	2364480	2160	960	6840

Table 1: Base resource utilization for 16 packet processors

Component	LUTs	Registers	BRAM	URAM	DSP
Single processor	4635 (0.4%)	3780 (0.2%)	24 (1.1%)	32 (3.3%)	0
Remaining (PR)	59526 (5.0%)	125100 (5.3%)	90 (4.2%)	32 (3.3%)	384 (5.6%)
Scheduler	9169 (0.8%)	14886 (0.6%)	0	0	0
Remaining	104847 (8.9%)	215514 (9.1%)	180 (8.3%)	96(10.0%)	648 (9.5%)
Single wrapper	3069 (0.3%)	3036 (0.1%)	0	0	0
CMAC	9111 (0.8%)	13800 (0.6%)	0	18 (1.9%)	0
PCIe	42663 (3.6%)	68479 (2.9%)	110 (5.1%)	32(3.3%)	0
Switching	54073 (4.6%)	69809 (3.0%)	36 (1.7%)	32(3.3%)	0
Complete design	179706 (15.2%)	223115 (9.4%)	363 (16.8%)	338 (35.2%)	0
VU9P device	1182240	2364480	2160	960	6840

Table 2: Base resource utilization for 8 packet processors

There are two main components for the incoming and outgoing interfaces. First, the physical Ethernet interfaces are connected via MAC modules and FIFOs, colored in green. Second, there are PCIe modules for connecting to the host for control, accessing host DRAM, and providing a virtual Ethernet interface, together colored in yellow. The main component of the PCIe module is a multi-queue PCIe DMA engine. This DMA subsystem includes a driver for the Linux networking stack, enabling operation as a NIC. These components are provided by the Corundum open-source 100 Gbps virtual network interface [6].

Finally, the switching subsystem is shown in orange, and each wrapper is connected to this subsystem; red lines show a few of these connections. Wider switches are implemented as 512-bits wide and the narrower switches as 128-bits wide, which provide max throughput of 128 Gbps and 32 Gbps respectively. There is some overhead for arbitration that reduces the performance from their maximum values, but remains above 100 Gbps for the wider interfaces.

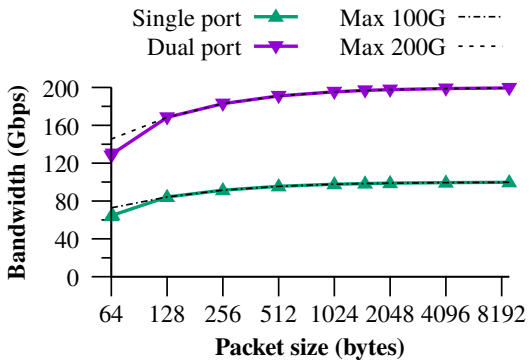
Tables 1 and 2 show the utilization breakdown of the 8 and 16 packet processor Shire runtimes. The main blocks, as well as the average resource utilization per packet processor, without any accelerators in them. The average remaining resources per PR block for each packet processor as well as the remaining resources in the reserved scheduler_block are also shown. Note that both tables show the Round Robin scheduler stats, and since for 8 packet processors the amount of arbitration logic decreases, we see less resource utilization and even Vivado decided not to use any Block RAMs. We are able to meet timing at 250 MHz for the design. The only hard IP blocks are the SERDES, PCIe and Gigabit CMAC; the rest are our own open source IP.

We developed an API library to provide the host-based control of packet processors described in this section. This library also integrates the Xilinx's PR-loading tool, `MCAP_tool`, and the Corundum 100 Gbps FPGA-to-host NIC driver [6]. Therefore, user can update a packet processor using the PR-loading, and initialize the packet processors memories and scheduler configurations through C code running on the host. We measured the time to reload a packet processor image using this library over 20 loads, and find that it takes 756 milliseconds on average.

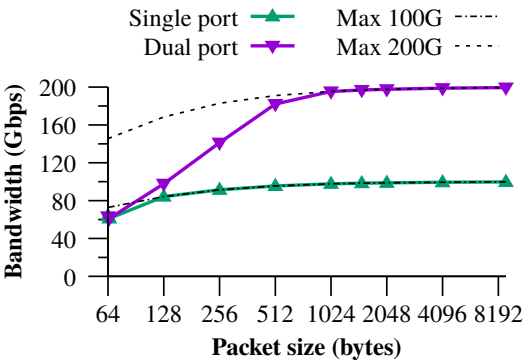
6 Evaluation

In this section we evaluate the performance of the framework through benchmarks to understand the limitations of different subsystems. We use a round robin scheduler for this evaluation, as well as optimized bare-metal code on the RISC-V to isolate user defined module overhead from the framework.

Our experiments are conducted using a host with an Intel(R) Xeon(R) CPU E3-1230 V2 running at 3.3 GHz and a PCIe Gen 3 x16 expansion bus. We install two separate Xilinx Virtex UltraScale+ FPGA VCU1525 boards into the PCIe bus, one serves as a traffic source/sink, while the other runs the system under test. Both ports of each FPGA are connected to the other FPGA with a 100-Gbps QSFP+ cable, so each can send and receive at 200 Gbps.



(a) 16 Packet processors



(b) 8 Packet processors

Figure 8: Full-throttle packet forwarding performance. Using (a) 16, and (b) 8 packet processing units.

6.1 Forwarding throughput

First we test the loss-free forwarding performance of the framework as a function of packet size. We consider packet sizes ranging from 64–8192 bytes by powers of two (excluding the 4-byte FCS), the worst case of 65 bytes, and typical datacenter MTU packet sizes of 1,500 and 9,000 bytes. we send packets from the tester FPGA at maximum rate on one or both interfaces and observe what portion of the packets make it back.

Figure 8 (a) and (b) shows the maximum forwarding rate as a function of packet size for 16 and 8 packet processors respectively. For 16 packet processors, this matches our packet generation capability except at the smallest packet sizes. (The dotted lines depict the maximum rate our traffic generator was able to source.) As shown, our 16 packet processor implementation can forward the offered load at 100 Gbps for every packet size other than 65 bytes, which achieves 89% of maximum rate. Our implementation can also forward the offered load at 200 Gbps with packets of least 128 bytes. 64- and 65-byte packets achieve lower rates of 88% and 89% of maximum rate respectively. For 8 packet processor units, we need at least 1024 byte packets to reach full 200 Gbps.

The performance drop for 64 and 65-byte packets for 16 packet processors comes mainly from the software latency.

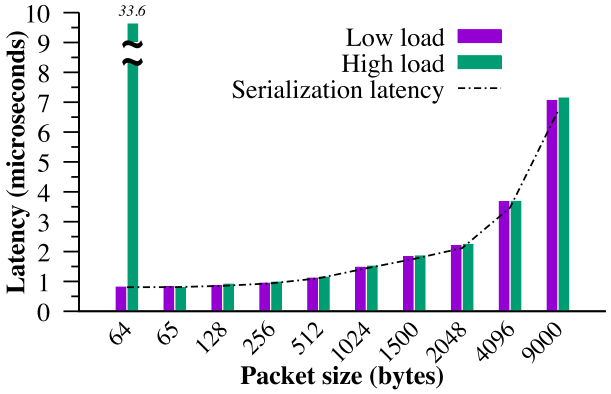


Figure 9: Round-trip latency as a function of packet size

The minimum time for our packet forwarder to read a descriptor, and send it back is 16 cycles. Therefore, each packet processor forwards a packet every 16 cycles, and with 16 packet processors we can hit at most 250 Mpps equal to the clock rate, for both packet sizes. Similarly, with 8 packet processor design we reach a maximum packet rate of 125 Mpps.

6.2 Forwarding latency

Next, we measure forwarding latency in terms of round trip time (RTT) from the traffic source, through the FPGA under test, and back to the original card. The packets are time-stamped just before leaving the core and latency is measured right upon arrival. Figure 9 shows the measured latencies for different packet sizes, under both low load and maximum load scenarios. The source of these delays are serialization. When a packet arrives at the FPGA and when it leaves the FPGA, we have to pay serialization latency at rate of 100 Gbps. Shire introduces another serialization latency at 32 Gbps, due to the fact that packet is fully loaded into each packet processor memory before the RISC-V core is notified, and also it has to be fully read out on the way out after the descriptor is released. The dotted line in Figure 9 shows the computed serialization delay according to Equation 1. Note that 0.765μsec is the minimum latency of packet going through our system.

$$Est.\,latency\,(\mu s) = (size * 8 * (\frac{2}{100} + \frac{2}{32})/1000) + 0.765 \quad (1)$$

High load introduces only marginal additional delay, which comes from packets not being perfectly uniform distributed between the two outgoing interfaces and have congestion. The only exception is 64-byte packets. The reason is that our packet generator can supply at full 200 Gbps, but our packet forwarder lacks behind. In steady state this results in receive FIFO becoming full, and hence the 35μsec latency.

6.3 Inter-core messaging performance

First, we measure the throughput of the inter-core loopback messaging system. Our implementation only uses a single 100-Gbps loopback port, as sending full packets among cores for each incoming packet is not the intended design. To test the performance of this loopback port, we implement a two-step forwarding system: we assign half of the cores to be recipients of the incoming traffic, and then each of these cores forwards packets to a specific core in the other half, and that core returns the packet to the link. We achieve 60% and 61% maximum throughput for packet sizes 64B and 65B respectively, mainly bottlenecked by the destination core header being attached to each packet. For packet sizes larger than 128 bytes the system can keep up with the 100-Gbps link rate.

Next, we perform two tests to measure the latency of broadcast messages, one with paced messages and one where each core is trying to send as many messages as it can. We timestamp each message and compare arrival time against transmit time. When paced, messages experience a latency between 72 to 92 ns. At full rate—which is not the intended use for this communication channel—we observe 1,596–1,680 ns of latency for the worse case design with 16 packet processors. This latency mostly comes from the 18 FIFO slots in each packet processor, 16 from actual FIFO and 2 from PR registers, which can be sent out every 16 cycles due to round-robin arbitration among cores, constituting 1,152 ns of this latency. Note that the write to the broadcast memory region will be blocked until there is room in the FIFO. The rest of the latency is due to FIFOs and registers in the switching subsystem and the software that can have slight variation.

7 Case Studies

In this section we evaluate how well Shire can reduce development effort and time, while achieving 200 Gbps line rate throughput. We demonstrate this with two case studies.

Case study 1: Porting Pigasus to achieve 200G. We describe how ported someone else's custom hardware accelerator it to Shire to scale up the performance. For this case study, we chose Zaho et al's Pigasus IDS [31], which is the first and open source hardware design to achieve 100 Gbps throughput. The main questions we answer in this case study is: can we easily port the core Pigasus hardware accelerators—string and port matching—to Shire's packet processors? How well can we implement packet reordering needed by IDSes in Shire? How much will we improve Pigasus's performance?

Case study 2: Building a Blacklisting Firewall. For our second case study, we wanted to evaluate how hard it is to make a new accelerator from scratch. We chose to implement simple firewall with a hardware accelerator that blocks packets that match an IP blacklist.

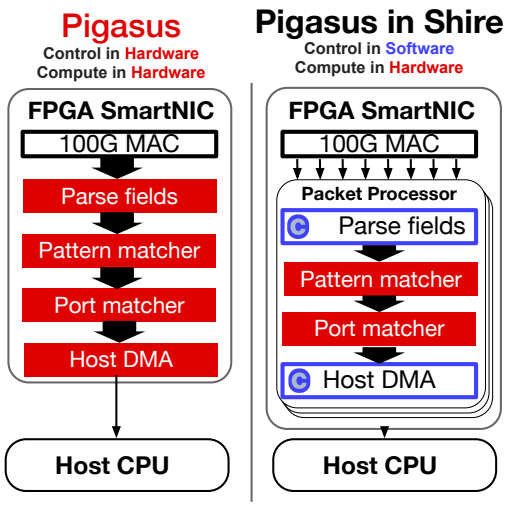


Figure 10: How we ported the Pigasus IDS to Shire.

We were able to develop both of these case studies in less than a month, and we improved the line rate of the Pigasus IDS from 100 Gbps to 200 Gbps for average internet packet size of 800B, and for packet size of 256B and above for firewall.

7.1 Pigasus IDS/IPS

7.1.1 Why is an IDS hard to develop in an FPGA?

IDSes identify suspicious behavior by monitoring network traffic and comparing it to a set of known fingerprints, stored in a constantly evolving ruleset. Many operators run all incoming traffic through an IDS, however they often have to divide traffic across clusters of servers to handle the computationally-intensive pattern matching for line-rate traffic [8]. Therefore, FPGAs are often considered for accelerating IDSes [4, 14, 26]. The Pigasus team built the first FPGA-based IDS accelerator to provide 100 Gbps acceleration for the Snort IDS running on a single server.

The most computationally expensive part of the processing in Pigasus is checking to see if traffic matches any of the rules in the ruleset—a task that is easy to parallelize. The Pigasus team had to build from scratch a significant fraction of their IDS hardware design to hit line rate on an FPGA. As shown in the left hand side of Figure 10, the developers had to build their own packet processing pipeline from scratch, including building hardware accelerators for parts of the processing that could be done in software, such as packet parsing.

7.1.2 Porting Pigasus to the Shire framework

Using the Shire framework, we can use the system architecture shown in the right hand side of Figure 10. We ported the two main hardware accelerators of the IDS—the pattern matcher and port matcher—to Shire. The DMA interface in the Shire packet processor was compatible with the streaming

packet interface of these accelerators, so we did not have to make significant changes to use them in Shire. Indeed, from the Pigasus code base we simply copied the files in the string pattern matcher and port matcher directories, and we added them to the Shire packet processor's top block.

We did make one small change to incorporate their accelerator to our design: because the RISC-V core communicates with the accelerator on demand, we added a FIFO for the input and output data of the accelerator, and its configuration registers, to be able to asynchronously control the accelerator from software running on the RISC-V core. Also we swapped a few components, such as FIFO modules, where they used Intel IP libraries, and we used our own Verilog libraries that were tested for Xilinx FPGAs. We also modified their rule packer module at the to output several chunks of 32 bits rather than 128 bits, matching our RISC-V word size. At this stage we were able to test the functionality of their accelerator within our Python testbench framework, running a basic C code to feed the accelerators with the incoming packets.

7.1.3 Porting challenges and new features

Next we tried to build an image from it for our FPGA. Unfortunately, the scaled up design for achieving 200 Gbps did not fit in our FPGA. After reaching out to the team, they mentioned that memory became a bottleneck for going to 200 Gbps, even when they used a large Intel Stratix 10 MX FPGA. However, upon a close look at the utilization report, we noticed that no large URAMs were used for the string or port matcher designs, while they had very large lookup tables. This is because URAMs cannot be initialized with tables when an FPGA bitstream is loaded, as they are targeted for FIFOs. One method would be to initialize them during runtime, but that would have required a separate development effort to enable initialization from the server hosting the FPGA.

Thanks to the memory subsystem in Shire, we were able to fill these tables at runtime. We simply added a write port to four of the memories in the matching accelerators.

We were able to fit about four times more instances of their accelerators in only 2/3rd of our similar-capacity FPGA (Table 3). Therefore, we used the same 250 MHz clock and we did not need to double-clock the logic at 400 MHz like the original design. In turn, this further simplified the design and reduced the required resources.

One caveat of the original Pigasus design is that there is no way to reconfigure the pattern matcher's ruleset during runtime: the only method to update the ruleset is to reload a new FPGA image. However, since several of these rulesets resided in the packet processor's memories, we could use the switching subsystem to modify them during runtime. Also, as it is native to the Shire design, we can use partial reconfiguration to change the smaller rulesets that do not use URAMs, or fix bugs in their accelerators at runtime.

To further evaluate the benefits and impacts of the Shire framework, we also wondered if we could implement TCP flow reordering in software on the RISC-V core instead of hardware. We built a hash-bashed packet scheduler for Shire to always send packets of the same flow to the same packet processor. In each packet processor, we used the scratch pad memory to keep 0.5 MB of flow state, to remember what is the time and sequence number of the last seen packet from each flow, tag of the stored flow to avoid collisions, and finally last 7 bytes required to be checked with the next packet. Note that we use 18 bit out of 32-bit hash result for index and the collision at the same time is very unlikely, as we also time out older flows. If we encounter reordering, we use up to half of our packet slots (e.g., 16) to buffer the out of order packets until the re-ordered packets arrive. Note that this solution will definitely be less efficient than the hardware based solution proposed in Pigasus, but use of software next to hardware can be used for prototyping or testing new ideas. Note that we could port the full Pigasus flow reassmebler into our scheduler, as we have 5 times more logic elements and 1.5 times more memory cells remaining when we built the basic RR scheduler (Table 2). Table 3 shows the average resource utilization break down inside each Pigasus packet processor unit, as well as the hash scheduler used for this mechanism.

Component	LUTs	Registers	BRAM	URAM	DSP
RISCV core	2008 (3.1%)	1015 (0.8%)	0	0	0
Mem. subsystem	3429 (5.3%)	896 (0.7%)	16 (14.0%)	32 (50.0%)	0
Accel. manager	1197 (0.1%)	2730 (0.1%)	0	0	0
Pigasus	35799 (55.8%)	49410 (38.3%)	56 (49.1%)	22 (34.4%)	80 (20.8%)
Total	42431 (66.1%)	54051 (41.9%)	72 (63.2%)	54 (84.4%)	80 (20.8%)
Packet processor	64161	128880	114	64	384
Scheduler	12233 (1.0%)	16499 (0.7%)	26 (1.2%)	0	0
Remaining	101783 (8.6%)	213901 (9.0%)	154 (7.1%)	96(10.0%)	648 (9.5%)

Table 3: Resource utilization for Packet processor with Pigasus, and Hash-based scheduler

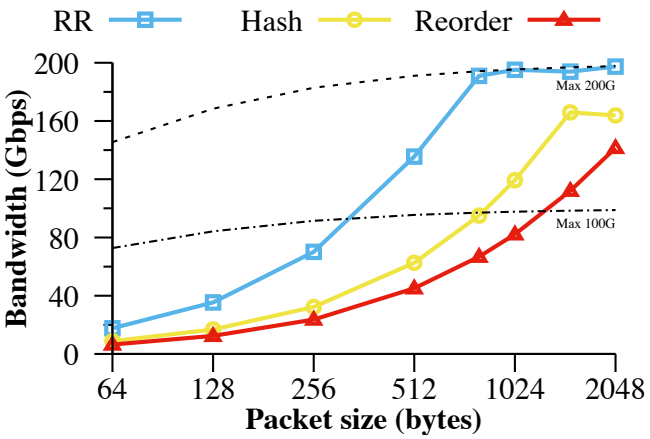


Figure 11: Pigasus performance in Shire

7.1.4 Evaluation

In our evaluation, we first made a packet trace based on the ruleset used for the generation of the Pigasus accelerator. After verifying that the expected number of packets are received

in hardware, we used tcpreplay to play back the attack pattern, and used another FPGA to fill the rest of the pipe with background traffic to hit 200 Gbps. Note that since attack packets are short we were not able to achieve more than 5.8 Gbps from tcpreplay, even using 16 concurrent tcpreplays, with full throttle mode, evenly distributed among the NUMA nodes. Figure 11 shows the result of this experiment.

The Round Robin (RR) line shows the use of an RR scheduler, this shows the best-case performance for Pigasus in Shire, assuming Pigasus reassmebly engine exists inside the scheduler. This baseline implementation achieved 200 Gbps for a larger packet size (>1024 Bytes), and it could hit 100 Gbps for a moderate packet size of 800 Bytes.

Next, we added a "Hash"-based scheduler in Shire alongside our software-based packet reordering. The scheduler pads the 4-byte hash result to the beginning of each packet. We observed that Shire could hit 166 Gbps for larger packet sizes. Note one reason for this drop is due to imperfect randomization of our packet generator, and hence uneven distribution of packets among the packet processors. Finally, On the "Reorder" line with 1% reordering, we see a drop to max rate of 143 Gbps. At a packet size of 800B, we can almost hit full 100 Gbps rate using the software reorder, as long as the reorder amount is less than the normal 0.3%.

It is worth noting that these results were achieved after we obtained a 30% improvement in packet rate by adjusting the order of members in an struct, and also applying a bug fix from the latest RISC-V GCC which was not available on the Arch Linux repositories. This suggests that there is room for compiler improvements, and also careful design of software is very important to achieve high performance in Shire.

7.2 Firewall

For our firewall case study, we built a simple firewall middlebox from scratch. The firewall checks every single packet, and if they have an IP that matches a blocklist the packet would be dropped, otherwise they are forwarded to the other Ethernet interface. In this case study, our goal was to evaluate how much effort is required to make even a simple middlebox in the Shire framework.

To implement a firewall, we built a simple IP prefix lookup accelerator from the list of 1050 blacklist IPs from the emerging threats[1]. We wrote a basic python script to generate a Verilog that first checks for the first 9 bits of the IP prefix, if they match, then it checks for the remaining 15 bits in the next cycle, and if there was a match it raises a flag in a register. We implemented the simple lookup this way so it could be performed in only two clock cycles.

Then we assigned a register address that the RISC-V core could use to load the IP into the accelerator, and another register to read the flag. The code below shows a small code snippet from the RISC-V code to show how this works, and the full code can be found in the Appendix. We load the IP address from the Ethernet packet using the DMA descriptor given by the switching subsystem. Then we load it into the IP matching accelerator (ACC_SRC_IP) and check the flag to see the results (ACC_FW_MATCH). Finally, if it was a match we drop the packet by setting the descriptor length to 0, and if it was not a match we forward it by swapping the port value between 0 and 1; this tells the switching subsystem to send the packet to the other 100 Gbps port.

[1] https://rules.emergingthreats.net/fwrules/emerging-PF-DROP.rules

Component	LUTs	Registers	BRAM	URAM	DSP
RISCV core	1896 (6.8%)	999 (1.8%)	0	0	0
Mem. subsystem	2183 (7.8%)	868 (1.6%)	16 (44.4%)	32 (100%)	0
Accel. manager	519 (0.0%)	1957 (0.1%)	0	0	0
Firewall IP checker	838 (3.0%)	197 (0.4%)	0	0	0
Total	5434 (19.5%)	4021 (7.2%)	16 (44.4%)	32 (100%)	0
packet processor	22405	55920	36	32	168

Table 4: packet processor resource utilization with Firewall

```
unsigned int src_ip =
    *((unsigned int*) (desc->data + 14 + 12));
ACC_SRC_IP = src_ip;
if (ACC_FW_MATCH) desc->len    = 0;
  else            desc->port ^= 1;
pkt_send(desc);
```

Doing similar packet-generator testing, we were able to hit at 200 Gbps, for sizes 256 bytes and above. Table 4 shows the average resource utilization break down inside each firewall packet processor unit.

8 Related work

There has been a significant amount of prior work in developing FPGA and System-on-Chip NIC frameworks that are flexible and debuggable. However, no prior framework has addressed the issues of required hardware parallelization for middleboxes, while providing the flexibility and debuggability of an C-language software, and the ability to change hardware acceleration—even during runtime—like we desire for an FPGA.

8.1 FPGA Frameworks

This work builds on a large body of prior work on improving the software and hardware development process for FPGAs, despite the fact that most of them were not targeted for middleboxes. As shown in Table 5, each of these prior frameworks demonstrated it was feasible to accomplish one or more of the goals of Shire [3, 5, 7, 9, 11–13, 19, 20, 22, 25, 32]. The goals we set out to accomplish were heavily inspired by these prior works: each of these frameworks demonstrated a new way that we could improve flexibility or debuggability, and several demonstrated a number of these features could be accomplished simultaneously.

Yet, none of these prior works demonstrated it was feasible to build a generic abstraction, like Shire, that can provide these

	Framework	SW Flexibility			HW Flexibility		Debuggability
		Prog. Model	*Statefulness*	*HW/SW interface*	*Partially Reconfig.*	*Add Custom Accel.*	*SW debuggable*
FPGA (Software)	**Shire**	C ✓	✓	✓	✓	✓	✓
	HxDP [3]	eBPF	✓	✓			✓
	RISC+NetFPGA [7]	C ✓	✓				✓
	AzureNIC [5]	Microcode	✓	✓		✓	
NIC SoCs (Software)	Cavium/Melanox SoCs	C ✓	✓	✓			✓
	PANIC [12]	C ✓				✓	
FPGA (Hardware)	Mellanox Innova NIC	Verilog				✓	
	NetFPGA [13]	Verilog				✓	
	Catapult [20]	Verilog				✓	
	P4FPGA [25]/SDNet	P4					
	FlowBlaze [19]	P4	✓				
	P4VBox [22]	P4			✓		
	ClickNP [11]	C (HLS)	✓			✓	
	Gorilla [9]	C (HLS)	✓			✓	

Table 5: A comparison of the development features of various programmable NIC frameworks.

features simultaneously in one FPGA development framework. For instance, the most recent framework HxDP [3], demonstrated a significant number of the development features could be provided by one framework. However, the HxDP framework focuses on software-based compilation development environment for FPGAs and does not provide an interface to add new hardware accelerators to their framework, and therefore it also does not provide a solution to parallelize custom accelerators. Other works' main focus was demonstrating that adding custom accelerators could be done in a well-defined framework on an FPGA. However, none of these works attempted to provide this at the same time as they provided software flexibility. [5, 9, 11, 13, 20]. In this work we demonstrate that it's not only feasible integrate hardware and software flexibility into one development abstraction, but it also demonstrates that there are significant benefits to do doing this. We also show that software flexibility benefits hardware flexibility.

8.2 Prior work that can benefit from Shire

There are numerous projects that implemented hardware accelerated network middleboxes with their own custom FPGA hardware pipeline, including an ML platform [18], a key value store [10], packet filtering [24], and several Intrusion Detection Systems [1, 4, 14, 26]. We believe that future efforts such as these may be bolstered by this platform, as developers will be able to focus on building their application-specific accelerators, and not have to manually tune a pipeline to get high-performance or manually build debugging hardware.

8.3 Hybrid SoC FPGA platforms.

A potential platform for our design is SoC-like FPGAs with hardened CPUs, such as Xilinx Zynq UltraScale+ MPSoC [30]. However, they have limited parallelism with a limited number of cores, small memory per core, low incoming bandwidth ($<$ 20 Gbps per core). Most importantly though, they have high communication latency to the logic fabric ($>$ 100 ns [27]) in the FPGA, and use a generic shared bus that introduces contention, both of which critically limit their ability to orchestrate parallel processing in the accelerators.

9 Conclusion and Discussion

We present Shire, a flexible and debuggable FPGA middlebox development framework. Shire provides a packet-processing abstraction consisting of a RISC-V core augmented with hardware accelerators, unified by a hybrid memory subsystem. Shire provides a packet distribution framework around these processing elements, and we demonstrate that it is possible to achieve 200 Gbps. We also demonstrate that Shire has a marginal effect on latency (especially when compared with PCIe and OS latencies). We plan to port Shire to several FPGA boards, from both Xilinx and Intel, to make it possible to use the same packet processors among them. We believe many applications, including research projects can benefit from this framework.

In the future, we believe that our packet processor-based framework can be "hardened" in an SoC architecture in future FPGAs. This will result in significantly faster packet processing performance that we can achieve instantiating RISC-V cores inside of an FPGA. Note that Xilinx has similar hardened IPs in their newest FPGAs, such as Network-on-Chip IP or AI Engines, which with minor modification can satisfy Shire needs. Finally, although FPGA-based middleboxes benefit most from the flexibility offered by Shire, we believe scope of Shire can be potentially wider. SoC-based SmartNIC designs might benefit from this framework to scale to higher link speeds. Fully custom ASIC designs can use Shire for their incremental builds where only the accelerators are updated between revisions.

References

[1] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski. OSNT: Open source network tester. *IEEE Network*, Sept. 2014.

[2] P. Biondi and the Scapy community. Scapy: Packet crafting for python. https://scapy.net/.

[3] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, T. Vergata, and R. Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[4] S. Campbell and J. Lee. Intrusion detection at 100G. In *Proc. The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2011.

[5] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[6] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[7] J. H. Han, N. Zilberman, B. A. Zeeb, A. Fiessler, and A. W. Moore. Prototyping RISC based, reconfigurable networking applications, 2016.

[8] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[9] M. Lavasani, L. Dennison, and D. Chiou. Compiling high throughput network processors. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012.

[10] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[11] B. Li, K. Tan, L. Luo, R. Luo, Y. Peng, N. Xu, Y. Xiong, and P. Cheng. ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM*, 2016.

[12] J. Lin, K. Patel, B. Stephens, S. Sivaraman, and A. Akella. PANIC: A programmable high-performance NIC for multi-tenant networks. In *Proc. USENIX OSDI*, 2020.

[13] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proc. IEEE International Conference on Microelectronic Systems Education (MSE)*, 2007.

[14] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[15] E. Networks. SD-WAN acceleration. https://ethernitynet.com/solutions/sd-wan-acceleration/.

[16] P. A. Networks. All release notes. https://docs.paloaltonetworks.com/release-notes.html.

[17] P. A. Networks. PA series next generation firewalls - hardware architectures - pa7000. https://www.paloaltonetworks.com/resources/pa-series-next-generation-firewalls-hardware-architectures.

[18] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *Proc. IEEE HotChips Symposium on High-Performance Chips (HotChips)*, 2015.

[19] S. Pontarelli et al. FlowBlaze: Stateful packet processing in hardware. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[20] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[21] K. Saban. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency.

[22] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 24, 2020.

[23] S. M. Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, 2015.

[24] J. C. Vega, M. A. Merlini, and P. Chow. Ffshark: A 100g fpga implementation of bpf filtering for wireshark. In *Proc. of the Field-Programmable Custom Computing Machines (FCCM) Symposium*, 2020.

[25] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *Proc. ACM Symposium on SDN Research (SOSR)*, 2017.

[26] N. Weaver, V. Paxson, and J. M. Gonzalez. The shunt: An FPGA-Based accelerator for network intrusion prevention. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2007.

[27] Xilinx. Minimum latency to access programmable logic register from ZYNQ ARM core. https://forums.xilinx.com/.

[28] Xilinx. UltraRAM: Breakthrough embedded memory integration on ultrascale+ devices, 2016.

[29] Xilinx. Vivado design suite user guide – heirarchical design, 2017.

[30] Xilinx. Zynq ultrascale+ device. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, 2019.

[31] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry. Achieving 100Gbps intrusion prevention on a single server. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[32] N. Zilberman, Y. Audzevich, G. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, Sept. 2014.

10 Appendix

The following code shows the C code running on RISC-V core for firewall case study.

```
#include "core.h"

// Accel wrapper registers mapping
#define ACC_SRC_IP    (*((volatile unsigned int
    *)(IO_EXT_BASE + 0x00)))
#define ACC_FW_MATCH (*((volatile unsigned char
    *)(IO_EXT_BASE + 0x04)))

static inline void slot_rx_packet(struct Desc*
    desc)
{
  unsigned short eth_type = *((unsigned short *)
      (desc->data + 12));
  unsigned int   src_ip   = *((unsigned int   *)
      (desc->data + 14 + 12));

  // check eth type
  if (eth_type == bswap_16(0x0800))
  {
    // start Firewall IP check
    ACC_SRC_IP = src_ip;
    if (ACC_FW_MATCH)
    {
      goto drop;
    }
    else
    {
      desc->port ^= 1;
      pkt_send(desc);
      return;
    }
  }

drop: //Non IPV4 or in firewall list
  desc->len = 0;
  pkt_send(desc);
}

int main(void)
{
  // Initializing scheduler and wrapper
  init_hdr_slots(16, 0x804000, 128);
  init_slots(16, 0x000000, 16384);

  // Enable only Evict and Poke Interrupts
 set_masks(0x30);

  while (1)
  {
    // check for new packets
    if (in_pkt_ready())
    {
      struct Desc desc;
      // read descriptor
      read_in_pkt(&desc);
      slot_rx_packet(&desc);
    }
  }

  return 1;
}
```