Azure Accelerated Networking: SmartNICs in the Public Cloud

Daniel Firestone Andrew Putnam Sambhrama Mundkur Derek Chiou Alireza Dabagh
Mike Andrewartha Hari Angepat Vivek Bhanu Adrian Caulfield Eric Chung
Harish Kumar Chandrappa Somesh Chaturmohta Matt Humphrey Jack Lavier Norman Lam
Fengfen Liu Kalin Ovtcharov Jitu Padhye Gautham Popuri Shachar Raindel Tejas Sapre
Mark Shaw Gabriel Silva Madhan Sivakumar Nisheeth Srivastava Anshuman Verma Qasim Zuhair
Deepak Bansal Doug Burger Kushagra Vaid David A. Maltz Albert Greenberg
Microsoft

Abstract

Modern cloud architectures rely on each server running its own networking stack to implement policies such as tunneling for virtual networks, security, and load balancing. However, these networking stacks are becoming increasingly complex as features are added and as network speeds increase. Running these stacks on CPU cores takes away processing power from VMs, increasing the cost of running cloud services, and adding latency and variability to network performance.

We present Azure Accelerated Networking (AccelNet), our solution for offloading host networking to hardware, using custom Azure SmartNICs based on FPGAs. We define the goals of AccelNet, including programmability comparable to software, and performance and efficiency comparable to hardware. We show that FPGAs are the best current platform for offloading our networking stack as ASICs do not provide sufficient programmability, and embedded CPU cores do not provide scalable performance, especially on single network flows.

Azure SmartNICs implementing AccelNet have been deployed on all new Azure servers since late 2015 in a fleet of $>$1M hosts. The AccelNet service has been available for Azure customers since 2016, providing consistent $<15\mu$s VM-VM TCP latencies and 32Gbps throughput, which we believe represents the fastest network available to customers in the public cloud. We present the design of AccelNet, including our hardware/software codesign model, performance results on key workloads, and experiences and lessons learned from developing and deploying AccelNet on FPGA-based Azure SmartNICs.

1 Introduction

The public cloud is the backbone behind a massive and rapidly growing percentage of online software services [1, 2, 3]. In the Microsoft Azure cloud alone, these services consume millions of processor cores, exabytes of storage, and petabytes of network bandwidth. Network performance, both bandwidth and latency, is critical to most cloud workloads, especially interactive customer-facing workloads.

As a large public cloud provider, Azure has built its cloud network on host-based software-defined networking (SDN) technologies, using them to implement almost all virtual networking features, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and access control lists (ACLs), virtual routing tables, bandwidth metering, QoS, and more. These features are the responsibility of the host platform, which typically means software running in the hypervisor.

The cost of providing these services continues to increase. In the span of only a few years, we increased networking speeds by 40x and more, from 1GbE to 40GbE+, and added countless new features. And while we built increasingly well-tuned and efficient host SDN packet processing capabilities, running this stack in software on the host requires additional CPU cycles. Burning CPUs for these services takes away from the processing power available to customer VMs, and increases the overall cost of providing cloud services.

Single Root I/O Virtualization (SR-IOV) [4, 5] has been proposed to reduce CPU utilization by allowing direct access to NIC hardware from the VM. However, this direct access would bypass the host SDN stack, making the NIC responsible for implementing all SDN policies. Since these policies change rapidly (weeks to months), we required a solution that could provide software-like programmability while providing hardware-like performance.

In this paper we present Azure Accelerated Networking (AccelNet), our host SDN stack implemented on the FPGA-based Azure SmartNIC. AccelNet provides near-native network performance in a virtualized environment, offloading packet processing from the host CPU to the Azure SmartNIC. Building upon the software-based VFP host SDN platform [6], and the hardware and software infrastructure of the Catapult program [7, 8], AccelNet provides the performance of dedicated hardware, with the programmability of software running in the hypervisor. Our goal is to present both our design and our experiences running AccelNet in production at scale, and lessons we learned.

2 Background

2.1 Traditional Host Network Processing

In the traditional device sharing model of a virtualized environment such as the public cloud, all network I/O to and from a physical device is exclusively performed in the host software partition of the hypervisor. Every packet

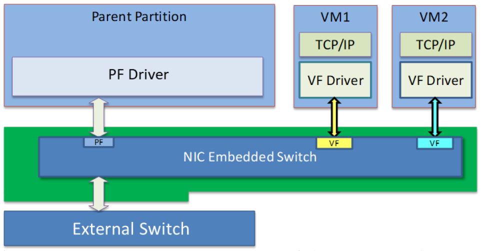


Figure 1: An SR-IOV NIC with a PF and VFs.

sent and received by a VM is processed by the Virtual Switch (vSwitch) in the host networking stack. Receiving packets typically involves the hypervisor copying each packet into a VM-visible buffer, simulating a soft interrupt to the VM, and then allowing the VM's OS stack to continue network processing. Sending packets is similar, but in the opposite order. Compared to a non-virtualized environment, this additional host processing: reduces performance, requires additional changes in privilege level, lowers throughput, increases latency and latency variability, and increases host CPU utilization.

2.2 Host SDN

In addition to selling VMs, cloud vendors selling Infrastructure-as-a-Service (IaaS) have to provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more. These semantics are sufficiently complex and change too frequently that it isn't feasible to implement them at scale in traditional switch hardware. Instead, these are implemented on each host in the vSwitch. This scales well with the number of servers, and allows the physical network to be simple, scalable and very fast.

The Virtual Filtering Platform (VFP) is our cloud-scale programmable vSwitch, providing scalable SDN policy for Azure. It is designed to handle the programmability needs of Azure's many SDN applications, providing a platform for multiple SDN controllers to plumb complex, stateful policy via match-action tables. Details about VFP and how it implements virtual networks in software in Azure can be found in [6].

2.3 SR-IOV

Many performance bottlenecks caused by doing packet processing in the hypervisor can be overcome by using hardware that supports SR-IOV. SR-IOV-compliant hardware provides a standards-based foundation for efficiently and securely sharing PCI Express (PCIe) device hardware among multiple VMs. The host connects to a privileged physical function (PF), while each virtual machine connects to its own virtual function (VF). A VF is exposed as a unique hardware device to each VM, allowing the VM direct access to the actual hardware, yet still isolating VM data from other VMs. As illustrated in Figure 1 , an SR-IOV NIC contains an embedded switch to forward packets to the right VF based on the MAC address. All data packets flow directly between the VM operating system and the VF, bypassing the host networking stack entirely. This provides improved throughput, reduced CPU utilization, lower latency, and improved scalability.

However, bypassing the hypervisor brings on a new set of challenges since it also bypasses all host SDN policy such as that implemented in VFP. Without additional mechanisms, these important functions cannot be performed as the packets are not processed by the SDN stack in the host.

2.4 Generic Flow Table Offload

One of AccelNet's goals was to find a way to make VFP's complex policy compatible with SR-IOV. The mechanism we use in VFP to enforce policy and filtering in an SR-IOV environment is called Generic Flow Tables (GFT). GFT is a match-action language that defines transformation and control operations on packets for one specific network flow. Conceptually, GFT is comprised of a single large table that has an entry for every active network flow on a host. GFT flows are defined based on the VFP unified flows (UF) definition, matching a unique source and destination L2/L3/L4 tuple, potentially across multiple layers of encapsulation, along with a header transposition (HT) action specifying how header fields are to be added/removed/changed.

Whenever the GFT table does not contain an entry for a network flow (such as when a new network flow is started), the flow can be vectored to the VFP software running on the host. VFP then processes all SDN rules for the first packet of a flow, using a just-in-time flow action compiler to create stateful exact-match rules for each UF (e.g. each TCP/UDP flow), and creating a composite action encompassing all of the programmed policies for that flow. VFP then populates the new entry in the GFT table and delivers the packet for processing.

Once the actions for a flow have been populated in the GFT table, every subsequent packet will be processed by the GFT hardware, providing the performance benefits of SR-IOV, but with full policy and filtering enforcement of VFP's software SDN stack.

3 Design Goals and Rationale

We defined the GFT model in 2013-2014, but there are numerous options for building a complete solution across hardware and software. We began with the following goals and constraints as we set out to build hardware offloads for host SDN:

1. Don't burn host CPU cores

Azure, like its competitors, sells VMs directly to customers as an IaaS offering, and competes on the price of those VMs. Our profitability in IaaS is the difference between the price a customer pays for a VM and what it costs us to host one. Since we have fixed costs per server, the best way to lower the cost of a VM is to pack more VMs onto each host server. Thus, most clouds typically deploy the largest number of CPU cores reasonably possible at a given generation of 2-socket (an economical and performant standard) blades. At the time of writing this paper, a physical core (2 hyperthreads) sells for \$0.10-0.11/hr[1], or a maximum potential revenue of around \$900/yr, and \$4500 over the lifetime of a server (servers typically last 3 to 5 years in our datacenters). Even considering that some fraction of cores are unsold at any time and that clouds typically offer customers a discount for committed capacity purchases, using even one physical core for host networking is quite expensive compared to dedicated hardware. Our business fundamentally relies on selling as many cores per host as possible to customer VMs, and so we will go to great lengths to minimize host overheads. Thus, running a high-speed SDN datapath using host CPU cores should be avoided.

2. Maintain host SDN programmability of VFP

VFP is highly programmable, including a multi-controller model, stateful flow processing, complex matching capabilities for large numbers of rules, complex rule-processing and match actions, and the ability to easily add new rules. This level of programmability was a key factor in Azure's ability to give customers highly configurable and feature-rich virtual networks, and enabling innovation with new virtual networking features over time. We did not want to sacrifice this programmability and flexibility for the performance of SR-IOV — in fact we wanted SDN controllers to continue targeting VFP without any knowledge that the policy was being offloaded. This would also maintain compatibility on host servers that do not have the necessary hardware for AccelNet.

Offloading every rule to hardware is neither feasible nor desirable, as it would either constrain SDN policy or require hardware logic to be updated every time a new rule was created. However, we concluded that offloading all rules is unnecessary. Most SDN policies do not change during the duration of the flow. So all policies can be enforced in VFP software on the first packet of a new TCP/UDP flow, after which the actions for that flow can be cached as an exact-match lookup. Even for short flows, we typically observe at least 7-10 packets including handshakes, so processing only the first packet in software still allows the majority to be offloaded (if the offload action is fast and efficient).

3. Achieve the latency, throughput, and utilization of SR-IOV hardware

Basic SR-IOV NICs set an initial bar for what is possible with hardware-virtualized networking — bypassing the host SDN stack and schedulers entirely to achieve low (and consistent) latency, high throughput, and no host CPU utilization. Offloading only exact match flows with associated actions allows for a tractable hardware design with the full performance of a native SR-IOV hardware solution on all but the first packet of each flow.

4. Support new SDN workloads and primitives over time

VFP continues to evolve, supporting new requirements and new policies, and AccelNet must be able to evolve along with VFP. We were, and continue to be, very wary of designs that locked us into a fixed set of flow actions. Not only does AccelNet need to support adding/changing actions, but the underlying platform should allow for new workloads that don't map neatly to a single exact-match table.

5. Rollout new functionality to the entire fleet

A corollary to the previous requirement, the AccelNet platform needed to enable frequent deployment of new functionality in the existing hardware fleet, not just on new servers. Customers should not have to move their existing deployments to new VM types to enable new features. Similarly, maintaining the same SDN functionality across hardware generations makes development, qualification, and deployment easier for us.

6. Provide high single-connection performance

From our experience with software-based SDN, we knew that network processing on a single CPU core generally cannot achieve peak bandwidth at 40Gb and higher. A good way to scale throughput past the limit of what one core can process is to break single connections into multiple parallel connections, utilizing multiple threads to spread load to multiple cores. However, spreading traffic across multiple connections requires substantial changes to customer applications. And even for the apps that implement multiple connections, we saw that many do not scale well over many flows because flows are often bursty — apps will dump large messages onto one flow while others remain idle.

An explicit goal of AccelNet is to allow applications to achieve near-peak bandwidths without parallelizing the network processing in their application.

7. Have a path to scale to 100GbE+

We designed AccelNet for a 2015 server generation that was going to deploy 40GbE widely. But we knew that the number of cores per server and the networking bandwidths would continue to increase in future generations, with speeds of 100GbE and above likely in the near future. We wanted a SmartNIC design that would continue to scale efficiently as network speeds and the number of VMs increase.

8. Retain Serviceability

VFP was designed to be completely serviceable in the background without losing any flow state, and supports live migration of all flow state with a VM being migrated. We wanted our SmartNIC software and hardware stack to have the same level of serviceability.

[1] Azure D v3 series or AWS EC2 m4 series instances, with price varying slightly by region

4 SmartNIC Hardware Design

4.1 Hardware Options

Based on the above goals, we proceeded to evaluate different hardware designs for our SmartNIC architecture.

Traditionally Microsoft worked with network ASIC vendors, such as Intel, Mellanox, Broadcom, and others, to implement offloads for host networking in Windows — for example TCP checksum and segmentation offloads in the 1990s [9], Receive-Side Scaling (RSS) [10] and Virtual Machine Queues (VMQ) [11] for multicore scalability in the 2000s, and more recently stateless offloads for NVGRE and VxLAN encapsulation for virtual networking scenarios for Azure in the 2010s [12]. In fact, GFT was originally designed to be implemented by ASIC vendors as an exact match-action table in conjunction with SR-IOV, and we shared early design ideas widely in the industry to see if vendors could meet our requirements. After time, our enthusiasm for this approach waned as no designs were emerging that could meet all of the design goals and constraints laid out in Section 3.

One major problem for SmartNIC vendors is that SR-IOV is an example of an all-or-nothing offload. If any needed SDN feature cannot be handled successfully in the SmartNIC, the SDN stack must revert to sending flows back through the software-based SDN stack, losing nearly all of the performance benefit of SR-IOV offload.

We saw four different possible directions emerge: ASICs, SoCs, FPGAs, and sticking with existing CPUs.

4.1.1 ASIC-based NICs

Custom ASIC designs for SDN processing provide the highest performance potential. However, they suffer from a lack of programmability and adaptability over time. In particular, the long time span between requirement specifications and the arrival of silicon was on the order of 1-2 years, and in that span requirements continued to change, making the new silicon already behind the software requirements. ASIC designs must continue to provide all functionality for the 5 year lifespan of a server (it's not feasible to retrofit most servers at our scale). All-or-nothing offloading means that the specifications for an ASIC design laid out today must meet all the SDN requirements for 7 years into the future.

ASIC vendors often add embedded CPU cores to handle new functionality. These cores can quickly become a performance bottleneck compared to rest of the NIC processing hardware. In addition, these cores can be expected to take an increasing burden of the processing over time as new functionality is added, exacerbating the performance bottleneck. These cores are also generally programmed via firmware updates to the NIC, which is handled by the ASIC vendors and slows the deployment of new features.

4.1.2 Multicore SoC-based NICs

Multicore SoC-based NICs use a sea of embedded CPU cores to process packets, trading some performance to provide substantially better programmability than ASIC designs. These designs became widely available in the 10GbE NIC generation. Some, like Cavium [13], used general purpose CPU cores (MIPS, later ARM64), while others, like Netronome [14] and Tilera, had specific cores for network processing. Within this space, we much preferred the general purpose SoCs — based on our evaluation that they were easier to program (you could take standard DPDK-style code and run it in a familiar Linux environment). To our surprise, these didn't have much of a drawback in performance compared to similar-generation ASIC designs.

However, at higher network speeds of 40GbE and above, the number of cores increases significantly. The on-chip network and schedulers to scatter and gather packets becomes increasingly complex and inefficient. We saw often 10 μs or more delays associated with getting packets into a core, processing the packet, and back out to the network — significantly higher latency than ASICs, and with significantly more variability. And stateful flows tend to be mapped to only one core/thread to prevent state sharing and out-of-order processing within a single flow. Thus individual network flow performance does not improve much because embedded CPUs are not increasing performance at the same pace as network bandwidths. This leads to the problem of developers having to spread their traffic across multiple flows, as discussed in Section 3, limiting the performance advantage of faster networks to only the most parallel workloads.

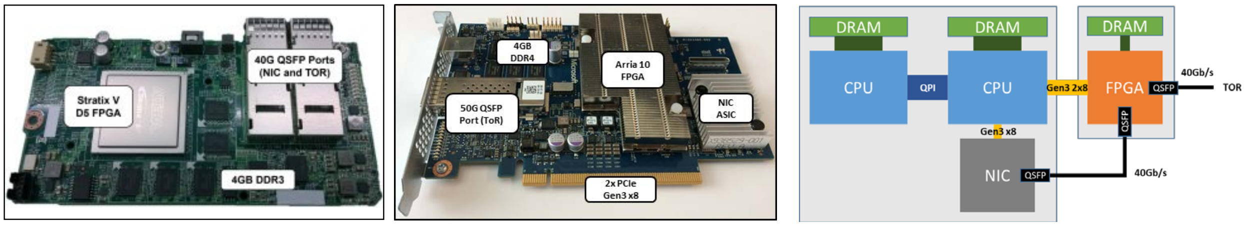
The future of SoC-style network offload is also questionable. At 10GbE, the total package was tolerable, with a few general purpose SoC cores being sufficient. 40GbE required nearly 4x the cores, though several vendors still created viable solutions. Still, 40GbE parts with software-based datapaths are already surprisingly large, power hungry, and expensive, and their scalability for 100GbE, 200GbE, and 400GbE looks bleak.

So while we found that the SoC approach has the advantage of a familiar programming model, the single-flow performance, higher latency, and poor scalability at higher network speeds left us looking for another solution.

4.1.3 FPGAs

Field programmable gate arrays (FPGAs) are reconfigurable hardware devices composed of small generic logic blocks and memories, all connected by a statically configured network. Programmers write code to assemble the generic logic and memory into ”soft logic” circuits, forming custom application-specific processing engines — balancing the performance of ASICs with the programmability of SoC NICs.

In contrast to CPUs like those on SoC-based NICs, an



(a) Azure SmartNIC Gen1, 40GbE w/ external NIC

(b) Azure SmartNIC Gen2, 50GbE w/ on-board NIC

(c) *bump-in-the-wire* architecture

Figure 2: Azure SmartNIC boards with Bump-in-the-Wire Architecture

FPGA is programmed only with the essential elements to complete the application, and can even take advantage of application characteristics such as the maximum size of the data to reduce bus widths and storage requirements. There are many studies that demonstrate FPGA can accelerate applications several orders of magnitude over pure software implementations for a wide range of application spaces from microprocessor simulation [15], genomics [16], machine learning [17], networking, pattern matching, graph processing, and so on.

The key characteristics of FPGAs that made it attractive for AccelNet were the programmability to adapt to new features, the performance and efficiency of customized hardware, and the ability to create deep processing pipelines, which improve single-flow performance.

When we evaluated SmartNIC options, Microsoft had already done the work to deploy FPGAs as datacenter accelerators for project Catapult [7] — we had a successful multi-thousand node cluster of networked FPGAs doing search ranking for Bing, with greatly-improved performance and lowered costs, and with a network transport layer running between the FPGAs within a rack. This led us to believe that FPGAs could be a viable option at scale for SmartNIC, as they had the potential to solve our dilemma of wanting the performance characteristics of an ASIC, but the programmability and reconfigurability inherent in a software solution like an SoC.

4.1.4 *Burn host cores*

We still evaluated all options against our original strategy of just using host cores to run our SDN stack, especially as technologies such as DPDK [18] showed that we could lower the cost of packet processing significantly by bypassing the OS networking stack and running cores in poll-mode. This option beat out ASICs given we couldn't get ASICs to meet our programmability requirements, but the cost and performance overhead of burning cores to our VM hosting costs was sufficiently high as outlined in Section 3 that even the inefficient multicore SoCs were a better approach.

4.2 Evaluating FPGAs as SmartNICs

FPGAs seemed like a great option from our initial analysis, but our host networking group, who had until then operated entirely as a software group, was initially skeptical — even though FPGAs are widely used in networking in routers, cellular applications, and appliances, they were not commonly used as NICs or in datacenter servers, and the team didn't have significant experience programming or using FPGAs in production settings. A number of questions below had to be answered before we decided to go down this path:

1. Aren't FPGAs much bigger than ASICs?

The generic logic portions of FPGAs are roughly 10x-20x bigger than identical logic in ASICs, since programmable memories (look up tables, or LUTs) are used instead of gates, and a programmable network of wires and muxes are used instead of dedicated wires to connect components together. If the FPGA design were simply generic logic, we should expect to need 10-20x more silicon area than an ASIC. However, FPGAs have numerous hardened blocks, such as embedded SRAMs, transceivers, and I/O protocol blocks, all of which are custom components nearly identical to those found in custom ASICs.

Looking at a modern NIC, the packet processing logic is not generally the largest part. Instead, size is usually dominated by SRAM memory (e.g. to hold flow contexts and packet buffers), transceivers to support I/O (40GbE, 50GbE, PCIe Gen3), and logic to drive these interfaces (MAC+PCS for Ethernet, PCIe controllers, DRAM controllers), all of which can be hard logic on an FPGA as well. Furthermore, modern ASIC designs often include significant extra logic and configurability (and even embedded CPU cores) to accommodate different requirements from different customers. This extra logic is needed to maximize volumes, handle changing requirements, and address inevitable bugs. Prior work demonstrates that such configurability can add an order of magnitude of area to ASIC logic [19]. So the trend has been for FPGAs to include more and more custom logic, while ASICs include more and more programmable logic, closing the efficiency gap between the two alternatives.

In practice, we believe for these applications FPGAs should be around 2-3x larger than similarly functioned ASICs, which we feel is reasonable for the massively increased programmability and configurability.

2. Aren't FPGAs very expensive?

While we cannot disclose vendor pricing publicly, the FPGA market is competitive (with 2 strong vendors), and we're able to purchase at significant volumes at our scale.

In our experience, our scale allows non-recoverable engineering costs to be amortized, and the cost of the silicon becomes dominated by the silicon area and yield. Total silicon area in a server tends to be dominated by CPUs, flash, and DRAM, and yields are typically good for FPGAs due to their regular structure.

3. Aren't FPGAs hard to program?

This question was the source of the most skepticism from the host networking team, who were not at the time in the business of digital logic design or Verilog programming. FPGAs can provide incredible performance compared to a CPU for programmable logic, but only if hardware designers really think through efficient pipeline designs for an application and lay them out as such. The project was originally assisted by the Catapult team in Microsoft Research, but eventually we built our own FPGA team in Azure Networking for SmartNIC. The team is much smaller than a typical ASIC design team, with the AccelNet team averaging fewer than 5 FPGA developers on the project at any given time.

Our experience on AccelNet as well as other projects within Microsoft, such as Bing Ranking [7, 8] for web search, LZ77 for data compression [20], and BrainWave [17] for machine learning, demonstrate that programming FPGAs is very tractable for production-scale cloud workloads. The exact same hardware was used in all four of these applications, showing the programmability and flexibility of the Azure SmartNIC expands well beyond SDN and network processing capabilities. This bodes well as we seek to add new functionality in years to come. Investment in strong development teams, infrastructure, simulation capabilities, and tools is essential, but much of this can be shared across different teams.

We have found the most important element to successfully programming FPGAs has been to have the hardware and software teams work together in one group, and use software development methodologies (e.g. Agile development) rather than hardware (e.g. Waterfall) models. The flexibility of the FPGA allows us to code, deploy, learn, and revise at a much faster interval than is possible for any other type of hardware desig. This hardware/software co-design model is what enables hardware performance with software-like flexibility.

4. Can FPGAs be deployed at hyperscale?

Getting FPGAs into our data centers was not an easy effort — when project Catapult started this was just not a common use case for FPGAs, and the team had to work through numerous technical, logistical, and team structure issues. However by the time we began SmartNIC, Catapult had worked out many of the common infrastructure details that were needed for a hyperscale deployment. The Catapult shell and associated software libraries abstracted away underlying hardware-specific details and allowed both hardware and software development for SmartNIC to focus largely on application functionality. Though much of this functionality is now common for FPGA vendors to support, at the time it wasn't. This project would not have been feasible without the prior Catapult work.

5. Isn't my code locked in to a single FPGA vendor?

Today's FPGA development is done almost entirely in hardware description languages like SystemVerilog (which we use), which are portable if the original development was done with the intention to facilitate porting. There are vendor-specific details, for example Intel FPGAs have 40b wide SRAMs versus Xilinx's 36b SRAMs, but once such details are accounted for, compiling code for a different FPGA is not that difficult. As a proof point of portability, project Catapult was first developed on Xilinx FPGAs, but was ported over to Altera FPGAs before our original pilot.

4.3 SmartNIC System Architecture

Even after selecting FPGAs as the path forward, there were still major questions about how to integrate it — where should the FPGA fit in our system architecture for our first SmartNIC, and which functions should it include? The original Catapult FPGA accelerator [7] was deliberately not attached the data center network to avoid being a component that could take down a server, and instead was connected over an in-rack backend torus network. This was not ideal for use in SDN offload, since the FPGA needed to be on the network path to implement VFP functionality.

Another option was to build a full NIC, including SR-IOV, inside the FPGA — but this would have been a significant undertaking (including getting drivers into all our VM SKUs), and would require us to implement unrelated functionality that our currently deployed NICs handle, such as RDMA[14]. Instead we decided to augment the current NIC functionality with an FPGA, and initially focus FPGA development only on the features needed for offload of SDN.

The converged architecture places the FPGA as a bump-in-the-wire between the NIC and the Top of Rack (TOR) switch, making the FPGA a filter on the network. A cable connects the NIC to the FPGA and another cable connects the FPGA to the TOR. The FPGA is also connected by 2 Gen3x8 PCIe connections to the CPUs, useful for accelerator workloads like AI and web search. When used as an accelerator, the network connection (along with an RDMA-like lossless transport layer using DCQCN [21]) allows scaling to workloads such as large DNN models that don't fit on one chip. The resulting first generation Azure SmartNIC, deployed in all Azure compute servers beginning in 2015, is shown in Figure 2(a).

The second generation of SmartNIC, running at 50GbE, Figure 2(b), is designed for the Azure Project Olympus OCP servers [22]. We integrated a standard NIC with SR-IOV on the same board as the FPGA, keeping the same bump-in-the-wire architecture, but eliminating the separate NIC board and the cable between the NIC and the

FPGA, reducing cost, and upgrading to a newer Intel Aria 10 FPGA.

5 AccelNet System Design

The control plane for AccelNet is largely unchanged from the original VFP design in [6], and remains almost entirely in the hypervisor. It remains responsible for the creation and deletion of flows from the flow table, along with determining the associated policy for each flow. The data plane for AccelNet is offloaded to the FPGA SmartNIC. The driver for the NIC is augmented with a filter driver called the GFT Lightweight Filter (LWF), which abstracts the details of the split NIC/FPGA hardware from VFP to make the SmartNIC appear as a single NIC with both full SR-IOV and GFT support, and to help in serviceability, as discussed in detail in Section 7.1.

5.1 Software Design

While the vast majority of the packet processing workload for GFT falls onto the FPGA hardware, software remains responsible for control operations, including the setup/teardown of flows, health monitoring, and enabling serviceability so that flows can continue during updates to VMs and the FPGA hardware. A high-level view of our architecture is shown in Figure 3. The flow table may not contain a matching rule for a given packet. In these cases, the offload hardware will send the packet to the software layer as an Exception Packet. Exception packets are most common on the first packet of a network flow, when the flow is just getting established.

A special virtual port (vPort) dedicated to the hypervisor is established for exception packets. When the FPGA receives an exception packet, it overloads the 802.1Q VLAN ID tag in the packet to specify that it is an exception path packet, and forwards the packet to the hypervisor vPort. VFP monitors this port and performs the necessary flow creation tasks after determining the appropriate policy for the packet's flow. If the exception packet was destined for a VM on the same host, the VFP software can deliver the packet directly to the VM. If the exception packet was outbound (sent by a VM for a remote destination), then the VFP software must resend the packet to the SmartNIC, which it can do using the same dedicated hypervisor vPort.

VFP also needs to be aware of terminated connections so that stale connection rules do not match to new network flows. When the FPGA detects termination packets such as TCP packets with SYN, RST or FIN flag set, it duplicates the packet — sending the original packet to its specified destination, and an identical copy of the packet to the dedicated hypervisor vPort. VFP uses this packet to track TCP state and delete rules from the flow table.

5.2 FPGA Pipeline Design

The GFT datapath design was implemented on the Azure SmartNIC hardware described in 4.3. For the remainder of this section we focus on the implementation on SmartNIC Gen1, though the same structure (with different values) applies to Gen2.

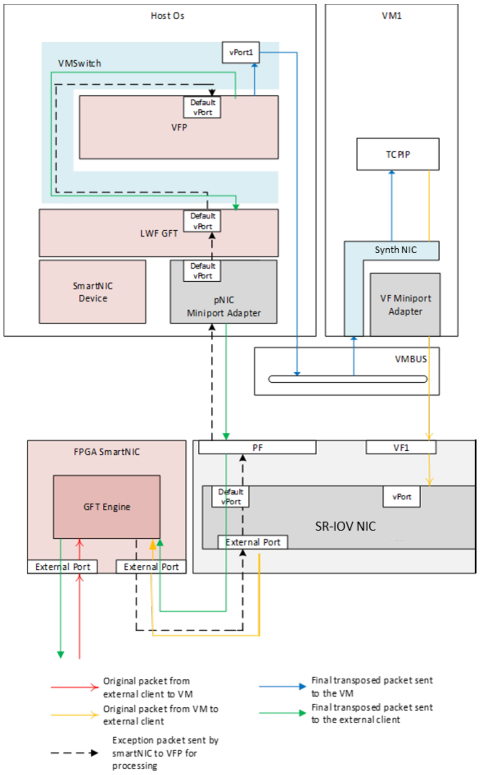


Figure 3: The SmartNIC GFT architecture, showing the flow of exception packets from the FPGA to software to establish a flow offloaded in hardware

The design of the GFT implementation on FPGA can be divided into 2 deeply pipelined packet processing units, each comprised of four major pipeline stages: (1) a store and forward packet buffer, (2) a parser, (3) a flow lookup and match, and (4) a flow action. A high-level view of our system architecture is shown in Figure 4.

The Parser stage parses the aggregated header information from each packet to determine its encapsulation type. We currently support parsing and acting on up to 3 groups of L2/L3/L4 headers (9 headers total, and 310 possible combinatons). The output of the parser is a key that is unique for each network flow.

The third processing stage is Match, which looks up the rules for the packet based on the unique key from the Parser stage. Matching computes the Toeplitz hash [23] of the key, and uses that as the cache index. We use a 2-level caching system with an L1 cache stored on-chip, and an L2 cache stored in FPGA-attached private DRAM.

The L1 cache is structured as a direct-mapped cache supporting 2,048 flows. We experimented with 2-way set associative caches and simpler hash algorithms than the Toeplitz hash [24], but found that the more computationally-intensive but less collision-prone Toeplitz hash coupled with the simpler direct-mapped

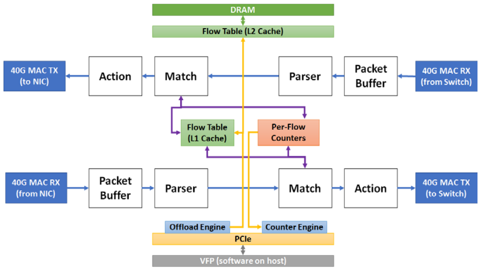


Figure 4: Block diagram of the GFT processing pipeline

cache resulted in better overall performance.

The L2 cache is structured as an 8-way set-associative cache, with support for O(1M) flows. Total number of supported flows is limited only by the DRAM capacity.

The final component is the Action stage, which takes the parameters looked up from the flow table, and then performs the specified transformations on the packet header. The Action block uses microcode to specify the exact behaviors of actions, enabling easy updates to actions without recompilation of the FPGA image. Only when entirely new actions are added will the FPGA need to be recompiled and a new bitstream loaded.

Non-trivial software-programmable Quality-of-Service guarantees can be implemented as optional components of the processing pipeline. For example, rate limiting can be done on a per-flow basis using a packet buffer in DRAM. Full description of our QoS frameworks and actions is beyond the scope of this paper. In total, the GFT role uses about 1/3 of the logic of the Intel Stratix V D5 chip that we used in the Gen1 SmartNICs.

5.3 Flow tracking and reconciliation

VFP is used by overlying controllers and monitoring layers to track per-flow connection state and data. GFT keeps track of all per-connection byte/flow counters, such as TCP sequence/ack numbers and connection state, and timestamps of the last time a flow got a packet. It periodically transmits all flow state to VFP via DMA transfers over PCIe, allowing VFP to ensure proper flow configurations, and to perform actions such as the cleanup of inactive flows.

GFT must also perform reconciliation so that flow actions get updated when VFP policy changes. Like VFP's unified flow table, GFT maintains a generation ID of the policy state on a system, and tracks what the generation ID when the rules for each flow were created. When a controller plumbs new policy to VFP, the generation ID on SmartNIC is incremented. Flows are then updated lazily by marking the first packet of each flow as an exception packet, and having VFP update the flow with the new policy actions.

6 Performance Results

Azure Accelerated Networking has been available since 2016. Performance results are measured on normal Azure AccelNet VMs in an Azure datacenter, running on Intel Xeon E5-2673 v4 (Broadwell at 2.3 Ghz) CPUs with 40Gbps Gen1 SmartNICs. Sender and receiver VMs are in the same datacenter and cross a Clos network of 5 standard switching ASICs between each other. We created no special configuration and the results, in Figure 5, should be reproduceable by any Azure customer using large Dv2 or Dv3 series Azure VMs. VFP policy applied to these VMs includes network virtualization, stateful NAT and stateful ACLs, metering, QoS, and more.

We measure one-way latency between two Windows Server 2016 VMs using registered I/O sockets [25] by sending 1 million 4-byte pings sequentially over active TCP connections and measuring response time. With our tuned software stack without AccelNet, we see an average of 50μs, with a P99 around 100μs and P99.9 around 300μs. With AccelNet, our average is 17μs with P99 of 25μs and P99.9 of 80μs — both latency and variance are much lower.

Azure offers VM sizes with up to 32Gbps of network capacity. With pairs of both Ubuntu 16.04 VMs and Windows 10 VMs with TCP congestion control set to CUBIC [26] and a 1500 Byte MTU, we consistently measure 31Gbps on a single connection between VMs with 0% associated CPU utilization in the host. Without AccelNet we see around 5Gbps on a single connection and multiple cores utilized in the host with enough connections running (~8) to achieve line rate. Because we don't have to scale across multiple cores, we believe we can scale single connection performance to 50/100Gb line rate using our current architecture as we continue to increase network speeds with ever-larger VMs.

For an example of a real world application, we deployed AccelNet to our Azure SQL DB fleet, which runs in VM instances, and ran SQL queries from an AccelNet VM in the same DC against an in-memory DB replicated across multiple nodes for high availability (both reads and writes to the service traverse multiple network hops). Average end-to-end read times went from ~1ms to ~300μs, and P99 read and write times dropped by over half as a result of reduced jitter in the network. Replication and seeding jobs that were often bound by the performance of a burst on a single connection ran over 2x faster.

Figure 6 shows comparative performance of AccelNet compared to other public cloud offerings that we measured on latest generation Intel Skylake-based instances (Fs72 v2 on Azure, c5.18xlarge on AWS, n1-highcpu-64 on GCP, measured in November 2017). All tests used unmodified Ubuntu 16.04 images provided by the platform with busy_poll enabled. We used the open source tools `sockperf` and `iperf` to measure latency and throughput, respectively. In our measurements, AccelNet had the lowest latencies, highest throughput, and lowest tail latencies (measured as percentiles of all pings over multiple 10-second runs of continuous ping-pong on established

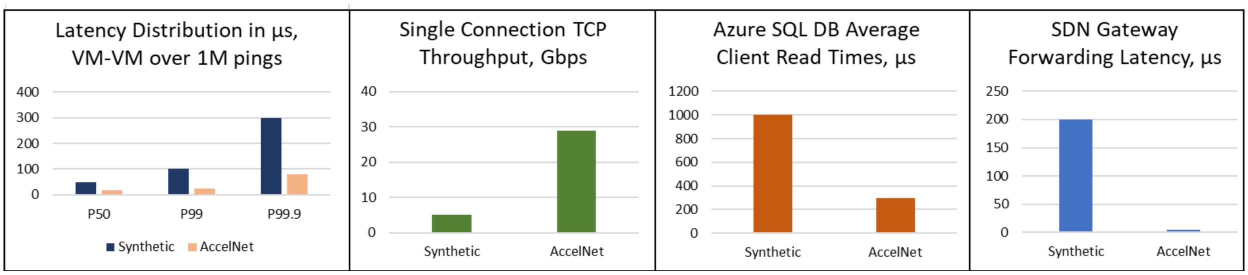


Figure 5: Selected performance data from Azure AccelNet VMs

TCP connections) of the instances we measured, including a consistent 10μs average latency between our Linux VMs. To test the performance of userspace I/O such as DPDK, we used a userspace TCP stack based on the open source VMA library [27], which achieved about 5μs latency pinging standard TCP sockets between our production VMs.

VFP is widely used in our fleet to run software gateways bridging between our SDN and external networks, such as ExpressRoute [28] circuits to customer datacenters. We used the programmable forwarding and QoS in our FPGA GFT implementation to offload all forwarding (after first packet on a flow) to the SmartNIC. Including encap/decap, stateful ACLs/NAT and metering, QoS, and forwarding, we saw a gateway forwarding line-rate 32Gbps traffic (even with just one flow), and consistent $<$5s latency with 0% host CPU utilization. Our prior gateway required multiple connections to hit line rate, burned CPU cores, and could spike to 100-200μs latency (including going to and from a VM) depending on system scheduling behavior. We believe this platform will let us create many more accelerated programmable appliances.

Using configurable power regulators on our SmartNIC, we've measured the power draw of our Gen1 board including all components in operational servers at 17-19W depending on traffic load. This is well below the 25W power allowed for a typical PCIe expansion slot, and on par with or less than other SmartNIC designs we've seen.

7 Operationalization

7.1 Serviceability

As with any other feature that is being built for the public cloud, serviceability, diagnostics and monitoring are key aspects of accelerated networking. The fact that both software and hardware are serviceable makes this particular scenario deployable for us. As discussed in [6], VFP is already fully serviceable while keeping existing TCP connections alive, and supports VM live migration with existing connections. With AccelNet, we needed to extend this serviceability as well — TCP flows and vNICs should survive FPGA reconfiguration, FPGA driver updates, NIC PF driver updates (which bring down VFs), and GFT driver updates.

We accomplished online serviceability by turning off hardware acceleration and switching back to synthetic vNICs to maintain connectivity when we want to service the SmartNICs or the software components that drive them, or when live migrating a VM. However, since AccelNet is exposed directly into the VM in the form of VFs, we must ensure that none of the applications break when the VF is revoked and the datapath is switched to synthetic mode. To satisfy this requirement, we do not expose the VF to the upper protocol stack in the VM directly. Instead, when the VF comes up, our synthetic NIC driver, the Hyper-V Network Virtual Service Consumer (NetVSC), marks the VF as its slave, either by using the slave mode present in the kernel for Linux VMs, or by binding to the NetVSC's upper NDIS edge in Windows VMs. We call this transparent bonding — the TCP/IP stack is bound only to the synthetic NIC. When VF is active, the synthetic adapter automatically issues sends over the VF adapter rather than sending it through the synthetic path to the host. For receives, the synthetic adapter forwards all receives from both the VF and synthetic path up the stack. When the VF is revoked for servicing, all transmit traffic switches to the synthetic path automatically, and the upper stack is not even aware of the VF's revocation or later reassignment. Figure 7 shows the accelerated data path and synthetic data path. The network stack is completely transparent to the current data path since NetVSC provides transparent bonding.

One of the benefits of SR-IOV is that VMs can use kernel bypass techniques like DPDK (Data Plane Development Kit) [18] or RDMA (Remote Direct Memory Access) to directly interface with hardware via the VF, but we needed to consider serviceability for them too when the VF is revoked, and the VM is potentially live-migrated elsewhere. We needed these applications to transparently fall back to a non-accelerated code path for that brief time period.

We found that there is no built-in fallback mechanism for many DPDK applications. So, we use a failsafe PMD (Poll Mode Driver) which acts as a bond between the VF PMD and a PMD on the synthetic interface. When the VF is active, the failsafe PMD operates over the VF PMD, thereby bypassing the VM kernel and the host software stack. When the VF is revoked for serviceability, the

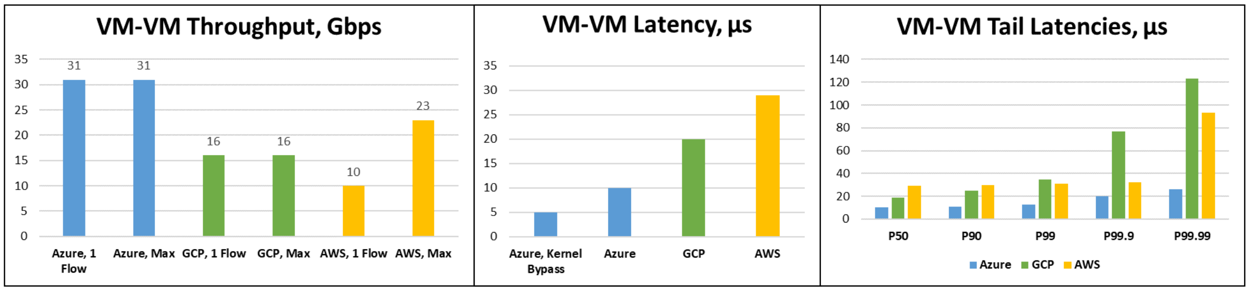


Figure 6: Performance of AccelNet VM-VM latencies vs. Amazon AWS Enhanced Networking and Google GCP Andromeda on Intel Skylake generation hardware.

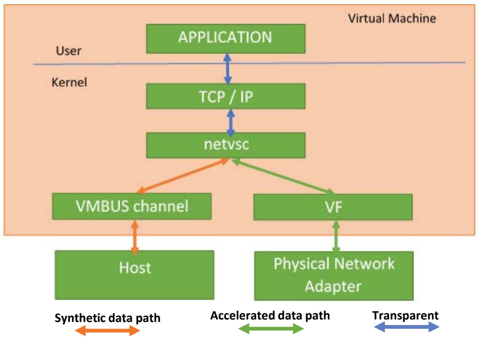


Figure 7: Transparent bonding between an SR-IOV interface and a synthetic interface

failsafe PMD starts operating over the synthetic path and packets flow through the VMBUS channels. Since the failsafe PMD exposes all DPDK APIs, the application does not see any difference except for a drop in performance for a short period of time.

For RDMA applications, this form of serviceability is harder and potentially involves many more queues. In practice, we found all our common RDMA applications are designed to gracefully fall back to TCP anyways, so we issue completions closing all RDMA queue pairs and let the app fail over to TCP. This isn't an issue for currently known workloads, but app-level transparency for RDMA serviceability remains an open question for the future if apps ever take a hard dependency on RDMA QPs staying alive.

Support for transparent VF bonding has been committed upstream in the Linux kernel (for NetVSC) and to dpdk.org for DPDK, and is natively available in Windows Server 2012 and later VMs. We've issued regular fleet-wide updates to all parts of the AccelNet stack, (VFP through GFT, the FPGA, and the PF driver), and found that transparent bonding works well in practice for our customers. While there is a short performance degradation while the synthetic path is active, apps stay alive and handle this well as they don't see a change in the network adapter they're bound to, or on active TCP connections. If an application is sensitive to this, we let VMs subscribe to an instance metadata service that sends notifications about upcoming maintenance and update events to the VM to enable it to prepare or move traffic elsewhere. If a VM is running behind the Azure load balancer, we can remove it from the active load balanced set during an update so that new external TCP/UDP flows are directed elsewhere for the duration of the update window.

7.2 Monitoring and Diagnostics

Monitoring is key to the reliability of a system like AccelNet at scale. Detection of early warning signs from both hardware and software and correcting them in an automated fashion is necessary to achieve production-quality reliability. We collect metrics from each component of AccelNet, including VFP, GFT, the SmartNIC and its driver, and the SR-IOV NIC and its driver — we have over 500 metrics per host (of which many are per-VM) collected in our scalable central monitoring system. Alerts and actions are triggered by combinations of these and we are constantly tweaking thresholds and actions as we get more experience and data over time with the system.

For diagnostics, we built programmable packet capture and tracing services at every interface on the SmartNIC — packet headers and data can be sampled at NIC/ToR ports on ingress/egress. We built a metadata interface along the network bus inside SmartNIC so that any module can emit diagnostic data about what exactly happened to a packet at that module, which is included in the capture. For example, in GFT we can trace how a packet was parsed, what flow it matched, what action it took, etc. We can collect hardware timestamps for these for accurate latency analysis. We also expose diagnostic information on key state machines as well as extensive counters, and automatically dump all critical internal state on an error.

8 Experiences

Azure SmartNICs and AccelNet have been deployed at scale for multiple years, with hundreds of thousands of customer VMs across our fleet. In our experience, AccelNet has improved network performance across the board for our customers without negatively impacting reliability or serviceability, and offering better throughput and latency than anything else we've measured in the public cloud. We believe our design accomplished all the goals we set out in Section 3 :

1. We stopped burning CPU cores to run the network datapath for AccelNet VMs. Host cores show less than 1% utilization used for exception processing.
2. SDN controllers have continued to add and program new policy in VFP, agnostic of the hardware offload now underneath.
3. We measured the overhead of the FPGA on latency as $<1\mu$s vs our SR-IOV NIC alone, and achieve line rate. This is much better than CPU cores alone.
4. We've continued to add new actions and primitives to GFT on the FPGA to support new workloads, as well as new QoS primitives and more.
5. Changes have been rolled out across multiple types of servers and SmartNIC hardware.
6. We can achieve line rate on a single connection.
7. We believe our design scales well to 100Gb+.
8. We have done production servicing of our FPGA image and drivers regularly for years, without negatively impacting VMs or applications.

8.1 Are FPGAs Datacenter-Ready?

One question we are often asked is if FPGAs are ready to serve as SmartNICs more broadly outside Microsoft. We certainly do not claim that FPGAs are always the best and only solution for accelerating networking in all cloud environments. The development effort for FPGA programming is certainly higher than software — though can be made quite tractable and agile with a talented hardware team and support from multiple stakeholders.

When Microsoft started Catapult, FPGAs were far from cloud-ready. Because SmartNIC shares a common development environment and history with Catapult, much of the development effort was shared across teams. We've observed that necessary tooling, basic IP blocks, and general support have dramatically improved over the last few years. But this would still be a daunting task for a new team. We didn't find that the higher level languages for FPGAs we experimented with produced efficient results for our designs, but our trained hardware developers had no trouble rapidly iterating on our SystemVerilog code.

The scale of Azure is large enough to justify the massive development efforts — we achieved a level of performance and efficiency simply not possible with CPUs, and programmability far beyond an ASIC, at a cost that was reasonable because of our volume. But we don't expect this to be a natural choice for anyone beyond a large-scale cloud vendor until the ecosystem evolves further.

8.2 Changes Made

As we expected, we continued adding all kinds of actions over time as the SDN stack evolved that we could never have predicted when we started, such as new stateful tunneling modes and state tracking. We believe responding to these rapidly shifting requirements would never have worked in an ASIC development flow. A small selection of examples include:

- We've repeatedly extended our TCP state machine with more precise seq/ack tracking of every TCP flow in our system for various functional and diagnostic purposes. For example, an ask to inject TCP resets into active flows based on idle timeouts and other parameters necessitated VFP being aware of the latest valid sequence numbers of every flow.
- We created a number of new packet forwarding and duplication actions, for example supporting tap interfaces with their own encapsulation and SDN policy, complex forwarding actions for offloading gateways and software routers to hardware, and multicast semantics using fast hardware replication on a unicast underlay.
- We added SDN actions such as NAT46 with custom translation logic for our internal workloads, support for virtualizing RDMA, and new overlay header formats.
- As we saw pressure to improve connection setup performance, we repeatedly iterated on our offload path, moving many functions such as hashing and table insertion of flows from software into hardware over time based on production telemetry.
- We've used the FPGAs to add constant new datapath diagnostics at line rate, including programmable packet captures, packet tracing through stages in our FPGA for latency and correctness analysis, and extensive counters and telemetry of the kind that require support in hardware in the datapath. This is our most constant source of iteration.

8.3 Lessons Learned

Since beginning the Azure Accelerated Networking project, we learned a number of other lessons of value:

- **Design for serviceability upfront**. The topics in Section 7 were the hardest of anything here to get right. They worked only because the entire system, from hardware to software to VM integration, were designed to be serviceable and monitorable from day 1. Serviceability cannot be bolted on later.
- **Use a unified development team**. If you want Hardware/Software co-design, hardware devs should in the same team as software devs. We explicitly built our hardware team inside the existing Azure host networking group, rather than the traditional approach of having separate groups for HW and SW, to encourage frequent collaboration and knowledge sharing.
- **Use software development techniques for FPGAs**. One thing that helped our agility was viewing the host networking datapath stack as a single stack and ship vehicle across VFP and FPGA, reducing complex rollout dependencies and schedule mismatch. As much as possible, we treated and shipped hardware logic as if it was software. Going through iterative rings of software qualification meant we didn't need ASIC-levels of specification and verification up front and we could be

more agile. A few minutes in a live environment covers far more time and more scenarios than typical RTL verification flows could ever hope to cover.

- **Better perf means better reliability**. One of the biggest benefits of AccelNet for VMs is that the network datapath no longer shares cores or resources with the rest of the host, and is not subject to transient issues — we've seen much more reliable performance and lower variance as a result.
- **HW/SW co-design is best when iterative**. ASIC development traditionally means designing a specification and test methodology for everything that a system could possibly want to do over its lifetime upfront. FPGAs allowed our hardware developers to be far more agile in their approach. We can deploy designs directly to customers, collect data from real workloads with detailed counters, and use those to decide what functions should be in hardware vs. software in the next release, and where performance bottlenecks are. More importantly, we can allow the specifications to evolve constantly throughout the development process. For example, we changed the hashing and caching strategies several times after the initial release.
- **Failure rates remain low**, and were in line with other passive parts in the system, with the most frequently failing part being DRAM. Overall the FPGAs were reliable in datacenters worldwide.
- **Upper layers should be agnostic of offloads**. Because VFP abstracted out whether SDN policy was being offloaded or not from controllers and upper layers, AccelNet was much less disruptive to deploy.
- **Mitigating Spectre performance impact**. In the wake of the Meltdown and Spectre attacks on our CPUs, CPU-based I/O was impacted by common mitigations [29]. Because AccelNet bypasses the host and CPUs entirely, our AccelNet customers saw significantly less impact to network performance, and many redeployed older tenants to AccelNet-capable hardware just to avoid these impacts.

9 Related Work

Since we first deployed Azure SmartNICs and announced them at the Open Networking Summit in 2015, we've seen numerous different programmable NIC solutions with vSwitch offload come to market (recently many of these are labeled as "Smart NICs" too). Most follow the trends we discussed in 4.1. Some [30] are based on ASICs with internal match-action tables — these tend to not be very flexible or support some of the more advanced actions we've implemented over time in GFT, and give little room for growth as actions and policy change. Others [13, 14] do datapath processing entirely in embedded cores, either general purpose CPUs or network-specific ones, but we've found the performance of this model is not great and we don't see a path to scale this to 100G and beyond without requiring many cores. A newer trend is to combine an ASIC supporting some match-action function with a small SoC supporting a DPDK-style datapath for on-core packet processing. But we don't ultimately see that this solves the dilemma of ASICs vs CPUs — if you have a widely-applied action that the ASIC can't handle, you have to send all your packets up to the CPUs, and now your CPUs have to handle line rate processing.

Others [31] show that they can improve the performance of software stacks entirely in the host and suggest burning cores to do host SDN. While we believe this in practice requires multiple cores at line rate for our workloads, in IaaS even taking a very small number of cores is too costly for this to make sense for us, and the performance and latency aren't optimal. With FPGAs, we've found we're able to achieve sufficient programmability and agility in practice. Offloading functionality to the switches as in [32] was also explored, but since we have to store complex actions for every TCP connection in our system, and with the increase of VM and container density on a node, we found the min set of policy needed to be offloaded, even when reasonably compressed, is at least 2 orders of magnitude more than even the latest programmable switch ASICs can store in SRAM - at NIC speeds we can scale out to GBs of DRAM.

Another recent suggestion is to use P4 engines [33], thus far mostly implemented in switches, to create SmartNICs. The P4 specification offers flexible parsing and relatively flexible actions, many of which are similar to GFT. In fact, P4 could potentially serve as a way to specify some of the GFT processing flow. However, there are other SDN functions outside the scope of existing P4 engines and even the P4 language specification that are important for us to implement in AccelNet — functions such as scheduling, QoS, background state updates, any kind of programmable transport layer, and a variety of complex policies outside the scope of simple packet transformations. While we expect the P4 language to be extended to include many of these, using a programmable fabric like an FPGA to implement GFT or P4 functionality remains a good choice given the evolving nature of the SDN and cloud space. We expect much of the functionality outside of the core packet processor to harden over time, but expect SDN to remain soft for the foreseeable future.

10 Conclusion and Future Work

We detailed the Azure SmartNIC, our FPGA-based programmable NIC, as well as Accelerated Networking, our service for high performance networking providing cloud-leading network performance, and described our experiences building and deploying them.

This paper describes primarily functions we were already doing in software in host SDN and offloaded to hardware for great performance. Future work will describe entirely new functionality we've found we can support now that we have programmable NICs on every host.

Acknowledgements

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers for their many helpful comments that greatly improved the final paper.

Azure Accelerated Networking represents the work of many engineers, architects, product managers, and leaders across Azure and Microsoft Research over several years, more than we can list here. We thank the entire Azure Networking, Server Infrastructure, and Compute teams for their support in developing, iterating on, and deploying SmartNICs and the Accelerated Networking service.

We thank Parveen Patel, Pankaj Garg, Peter Carlin, Tomas Talius, Jurgen Thomas, and Hemant Kumar for their invaluable early feedback in deploying preview versions of our service, and KY Srinivasan, Stephen Hemminger, Josh Poulson, Simon Xiao, and Haiyang Zhang for supporting our Linux VM ecosystem's move to AccelNet. Finally, in addition to our leaders in the author list, we thank Yousef Khalidi, Mark Russinovich, and Jason Zander for their support.

References

[1] Microsoft Azure. `http://azure.microsoft.com`, 2018.

[2] Amazon. Amazon Web Services. `http://aws.amazon.com`, 2018.

[3] Google. Google Cloud Platform. `http://cloud.google.com`, 2018.

[4] Microsoft. Overview of Single Root I/O Virtualization (SR-IOV). `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-`, Apr 2017.

[5] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–10, Jan 2010.

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

[7] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Gopi Prashanth, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[8] 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)*, pages 1–13, Oct 2016.

[9] Microsoft. TCP/IP Offload. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/tcp-ip-offload`, Apr 2017.

[10] Microsoft. Introduction to Receive Side Scaling. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling`, Apr 2017.

[11] Microsoft. Virtual Machine Queue (VMQ). `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/virtual-machine-queue--vmq-`, Apr 2017.

[12] Microsoft. Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/network-virtualization-using-generic-routing-encapsulation--nvgre--task-offload`, Apr 2017.

[13] Cavium. Cavium LiquidIO II Network Appliance Smart NICs. `http://www.cavium.com/LiquidIO-II_Network_Appliance_Adapters.html`.

[14] Netronome. Open vSwitch Offload and Acceleration with Agilio CX SmartNICs. `https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf`.

[15] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Yatish Turakhia, Kevin Jie Zheng, Gill Bejerano, and William J. Dally. Darwin: A hardware-acceleration framework for genomic sequence alignment. *bioRxiv*, 2017.

[17] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, and Doug Burger. Accelerating Persistent Neural Networks at Datacenter Scale. In *Hot Chips 27*, 2017.

[18] DPDK. DPDK: Data Plane Development Kit. `http://dpdk.org/about`, 2018.

[19] Gokhan Sayilar and Derek Chiou. Cryptoraptor: High throughput reconfigurable cryptographic processor. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 154–161, Piscataway, NJ, USA, 2014. IEEE Press.

[20] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, May 2015.

[21] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.

[22] Microsoft. Server/ProjectOlympus. `www.opencompute.org/wiki/Server/ProjectOlympus`, 2018.

[23] P. P. Deepthi and P. S. Sathidevi. Design, implementation and analysis of hardware efficient stream ciphers using lfsr based hash functions. *Comput. Secur.*, 28(3-4):229–241, May 2009.

[24] Microsoft. RSS Hashing Functions. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/rss-hashing-functions`, Apr 2017.

[25] Microsoft. Registered Input/Output (RIO) API Extensions. `https://technet.microsoft.com/en-us/library/hh997032(v=ws.11).aspx`, Aug 2016.

[26] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018.

[27] Messaging Accelerator (VMA). `https://github.com/Mellanox/libvma`, 2018.

[28] Microsoft. ExpressRoute overview. `https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction`, Oct 2017.

[29] Microsoft. Securing Azure customers from CPU vulnerability. `https://azure.microsoft.com/en-us/blog/securing-azure-customers-from-cpu-vulnerability/`, 2018.

[30] Chloe Jian Ma and Erez Cohen. OpenStack and OVS: From Love-Hate Relationship to Match Made in Heaven. `https://events.static.linuxfound.org/sites/events/files/slides/Mellanox%20OPNFV%20Presentation%20on%20OVS%20Offload%20Nov%2012th%202015.pdf`, Nov 2012.

[31] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

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

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.