Performance Evaluation of VirtIO Device Drivers for Host-FPGA PCIe Communication

Sahan Bandara*‡, Ahmed Sanaullah†, Zaid Tahir *, Ulrich Drepper†, and Martin Herbordt*‡
*ECE Department, Boston University †Red Hat Inc. ‡{sahanb, herbordt}@bu.edu

***Abstract*—Unleashing the full potential of FPGAs as offload devices requires Host-FPGA connectivity that is reliable, robust, and uniform; and that implements (at least) a required set of features and protocols. PCIe is the most widely used host-FPGA interface for high-performance applications. However, existing frameworks for host-FPGA PCIe communication have several limitations, including lack of portability and poor upstream support. Native VirtIO drivers in the host operating system can address many of these limitations on the host side. A complete VirtIO-based solution, however, also requires new support on the device side. In previous work, a general framework was proposed that requires little additional programming effort per new device. Although VirtIO drivers could provide an attractive alternative to vendor-provided device drivers, their performance when interacting directly with physical devices has not been explored.**

Given that VirtIO drivers are designed targeting *virtual* devices, it is critical to investigate whether they perform at an acceptable level when handling *physical* devices. In this work, we compare the performance of VirtIO device drivers to vendor-provided device drivers in terms of communication latency and latency distribution and show VirtIO drivers provide similar or slightly improved performance with reduced variance. To facilitate our analysis, we also extend prior work in implementing support for VirtIO network devices on FPGAs. The overall significance of this work is that it demonstrates the feasibility of replacing the vast space of legacy device drivers with the VirtIO drivers already native to the host OS.

I. Introduction

FPGAs are used as complexity offload devices for CPUs. Their inherent flexibility, combined with tight coupling of communication and computation, allow FPGAs to be used in a vast number of use cases where they are preferred to other accelerators such as GPUs. For instance, FPGAs are used to accelerate user and system applications, implement networking functions to process data at line rates, perform system administration in clouds, and provide secure enclaves with hardware isolation, and a myriad of other tasks [1]–[8]. To unleash the full potential of FPGAs as complexity offload devices, however, the host-FPGA connectivity must provide a reliable, robust, and uniform interface and implement a required set of features and protocols.

Typically, high-end FPGA devices targeting high-performance applications use PCIe [9] for host-FPGA connectivity. There are several limitations in existing frameworks: They are almost always vendor- and device-specific and lack portability. On the device side, the lack of portability stems from the use of vendor-provided IP cores and the underlying hardened ASIC blocks that implement the PCIe physical and link layer functions. FPGA devices from different vendors, or even different device families from the same vendor, may use different ASIC components; as a consequence, there are often inconsistencies in the features supported and the APIs exposed to user logic. What this means for the FPGA developer is that a design targeting a particular device cannot be ported to a different device without incurring significant engineering overhead.

On the host side, lack of portability and (invariably) poor maintenance are major limitations. These difficulties in maintaining device drivers for FPGAs stem from the lack of generic device drivers and the large space of drivers created by product developers and end users. The variations in capabilities and functionality across devices force the device drivers also to be device-specific. While this is an issue for all device drivers, FPGAs differ from other accelerators due to their flexibility: this adds another dimension to the already large space of FPGA device drivers. Since the same device can be used to implement applications with drastically different semantics, and deployed in different contexts, different device drivers are designed to accommodate these variations. For instance, a GPU is always used as a GPU and the interlocutor does not need to interact with the device as something other than a GPU. This allows a GPU vendor to provide generic device drivers to support all of their products.

In contrast, the same FPGA could be used to implement a Cryptographic accelerator, a storage accelerator, a SmartNIC, or a myriad of other applications, each with its own semantics. This flexibility makes it *impossible* for FPGA vendors to provide device drivers that can accommodate all potential use cases. Thus, FPGA vendors typically provide reference drivers for use as is, or as the starting point for a custom driver that satisfies the specific user requirements. The alternative to writing a new driver is to lift the device semantics to the application level, which is more likely to be sub-optimal.

With this vast space of device drivers, and most of the work on device drivers being done downstream, maintaining them becomes the responsibility of the FPGA designers or end users. Device drivers need to be updated whenever the kernel APIs used are updated. New mainline Linux kernels are released every 9-10 weeks [10]. Red Hat Enterprise Linux OS follows a 6-month release cycle [11]. Other popular Linux operating systems such as Ubuntu and Fedora follow similar release cycles. While not every kernel update or OS release may require driver updates, there is still that possibility.

As a concrete example, the XDMA device driver [12], used in the experimental setup of this work, has had 71 lines of code changed during the last year to support kernel updates.

These changes were made as a single commit to the repository. However, we have updated the XDMA driver three times in our testing during the last 1.5 years. This highlights how the updates to vendor-provided drivers are lagging behind kernel updates. In the case of large OEMs and cloud service providers with their own drivers, a dedicated team is typically deployed; for smaller ones, maintenance and updates invariably lag. Both cases are extremely costly for some combination of maintainers, developers, and end users.

The use of generic device drivers can significantly reduce the space of custom FPGA device drivers for most use cases and so reduce the maintenance overhead. VirtIO [13] is an industry standard for I/O virtualization and is one possible solution to the challenges posed by the use of vendor-provided or user-developed device drivers. VirtIO is an abstraction layer over a host's devices for virtual machines running in a paravirtualized hypervisor. VirtIO drivers access the host's devices via minimal virtual devices called VirtIO devices. VirtIO devices only implement the bare necessities to enable sending and receiving data. They represent generic device types such as block devices, network adaptors, and consoles, which differ from fully emulated devices where the details of the physical device are replicated in software. In this work we investigate *repurposing* VirtIO for actual physical devices.

Since there is native support for VirtIO in common host operating systems—such as the Linux kernel—no additional drivers need to be written/maintained, and APIs are mostly consistent. VirtIO also supports feature negotiation, i.e., the device and driver can use feature bits to determine the subset of supported features to ensure compatibility. Moreover, there are additional benefits of exposing the FPGA to the host as a VirtIO device. For instance, it can reduce data copies and latency in a virtualized environment through direct communication between the VirtIO driver running in guest kernel space and the physical device, bypassing the host OS. Another major benefit of using VirtIO device drivers with FPGAs is the ability to use different device drivers, each with semantics matching the type of accelerator implemented on the FPGA. This also allows leveraging the operating system's software stack for certain common tasks instead of using the device driver or the user-level application to do so. This is crucial since FPGAs can be used to implement a large variety of functions, each with its own semantics.

A previous study [14] demonstrated that it is possible to use unmodified VirtIO drivers to communicate with FPGAs, and provided a description of how to implement a VirtIO-compliant interface on FPGAs. What is missing, however, is a performance comparison of VirtIO drivers versus legacy FPGA device drivers. This work remedies this lack with results at two levels: first, standalone measurements, such as latency for data transfers, and second, through real application performance. The specific contributions of this work are as follows:

- Added support for more VirtIO device types.
- Compare the performance of VirtIO and vendor-provided device drivers using round-trip average and tail latencies.
- Highlight the differences in device driver design, device/application semantics, and work allocation between software and hardware that impact the driver performance and our analysis.
- Demonstrate that replacing legacy drivers with VirtIO in no case results in reduced performance, but rather can even be beneficial, and often reduces variance.

The overall contribution is to demonstrate the viability of VirtIO for Host-FPGA PCIe communication.

II. Background

A. VirtIO Device Drivers

VirtIO devices are virtual devices found in virtual environments. However, they appear as physical devices to a guest within a virtual machine. According to the VirtIO specification, the purpose of VirtIO is to "provide a straightforward, efficient, standard, extensible mechanism for virtual devices, rather than boutique per-environment or per-OS mechanisms" [13]. VirtIO devices therefore: (i) use normal bus mechanisms for interrupts, DMA, etc., which are familiar to device driver authors; (ii) use rings of descriptors for input and output, which are carefully laid out to avoid effects from both the device and the driver writing to the same cache lines; (iii) make no assumptions regarding the environment they operate in, beyond the type of bus to which a device is connected; and (iv) include feature bits that allow the device and the operating system to negotiate features supported and used, enabling forward and backward compatibility.

The most basic VirtIO use-model is where an application executing in the guest user space uses the VirtIO front-end driver in the guest kernel space to interact with a virtual device emulated by a host user-space application. The front-end driver and the back-end device use queues named *virtqueues* to communicate with each other. In paravirtualization, where there is a physical device attached to the host machine, the guest application can use VirtIO drivers. Here, a device-specific legacy device driver runs in the host kernel space to allow communication with the physical device. Additional software is used to convert requests from the virtual back-end device to the semantics of the legacy device driver. Figure 1 depicts the typical paravirtualization setup and how a VirtIO-compliant interface on the FPGA can eliminate the need for emulated backend VirtIO devices and vendor-provided (or user-developed) device drivers specific to the given device.

B. Legacy Device Drivers

Due to significant differences among different FPGAs, device drivers for FPGAs are specific to vendors and device families. FPGA vendors provide reference device drivers compatible with different device families using similar PCIe IPs. For example, Xilinx provides two DMA IP reference drivers [12] where the XDMA driver supports Xilinx UltraScale+, UltraScale, Virtex-7 XT, and 7 Series Gen2 devices, while the QDMA driver supports UltraScale+ devices. End users can modify these reference drivers to match the specific requirements of their designs.

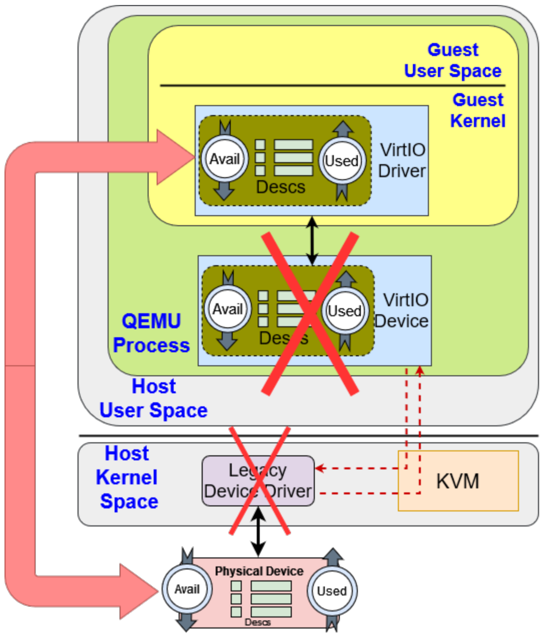

Fig. 1. VirtIO interface on the FPGA eliminates the need for back-end VirtIO devices and legacy device drivers.

FPGA vendors also provide runtime libraries, such as Xilinx runtime library (XRT) [15] and Open Programmable Acceleration Engine (OPAE) [16]. These provide simple APIs for programming, data movement, and controlling the FPGA. These runtime libraries are accompanied by FPGA shells and kernel-space device drivers written to match the PCIe IPs used in the FPGA shells. *These typically support a very limited set of high-end FPGA devices and lack portability even across devices from the same vendor.*

C. Using VirtIO Drivers to Interact with Physical Devices

VirtIO device drivers are designed with virtual devices in mind. But they use regular bus mechanisms to interact with VirtIO devices. As a consequence, VirtIO device drivers cannot differentiate between a virtual device and a physical device as long as the physical device presents a VirtIO-compliant interface. To do so, there are three main requirements: (i) Announce the correct device and vendor IDs at the time of device discovery and PCIe bus enumeration; (ii) Implement VirtIO configuration structures used for device initialization and operation; and (iii) Add the VirtIO capabilities to the device capability list.

The VirtIO configuration structures are implemented as part of the control logic on the FPGA and are mapped to one of the base address registers (BAR) of the device. The VirtIO capabilities added to the device capability list help the device driver locate the corresponding configuration structures. Achieving items (i) and (iii) may require modifications to the vendor-provided PCIe IPs. Descriptions of the controller implementation, modifications to the PCIe IP, and alternative implementation choices are provided in [14].

D. Related Work

The authors of [17] use VirtIO as the front-end driver to decrease communication latency between software and hardware when deploying multi-tenant FPGAs in Linux-based cloud infrastructure. An FPGA virtualization framework where VirtIO drivers are used as the front-end drivers is presented in [18]. In both studies, the VirtIO drivers are not communicating directly with the FPGAs; rather, a legacy device driver in the host kernel space is used to communicate with the FPGA over PCIe.

The only vendor-provided PCIe IP core with VirtIO support of which we are aware is the P-Tile Avalon Streaming Intel FPGA IP for PCIe [19]. This IP allows each of the PCIe physical functions (PF) and virtual functions (VF) implemented on the FPGA to have its own VirtIO configuration structures. A soft IP implementing the VirtIO capability for PFs and VFs is instantiated as a sub-IP when VirtIO is enabled; it also adds the required VirtIO capabilities to the device capability list.

The Silicom C5010X Data Center FPGA IPU NIC [20] is based on an Intel Stratix 10 DX FPGA. It can be deployed as a VirtIO network or storage accelerator. Details are not available regarding the IP cores used. However, the Intel P-Tile Avalon Streaming PCIe IP core discussed previously only supports Stratix 10 DX and Intel Agilex FPGAs. Therefore, the same PCIe IP may be used in this product.

A custom PCIe IP is used in [21] to implement VirtIO support on an FPGA PCIe endpoint. In contrast, the vendor-provided PCIe IP is modified in [14] to implement a VirtIO-compliant interface on the FPGA and allow unmodified VirtIO drivers to directly communicate with an FPGA. However, only a VirtIO console device is implemented, and the performance of VirtIO drivers is not explored.

There is a collection of prior work [22]–[29] focused on enhancing the usability of FPGAs through FPGA shells, implementing operating system-like abstractions on FPGAs, automated composition of systems from custom processing elements, FPGA virtualization and integrating into the host operating systems thread/process abstraction, etc. However, these works do not focus on the portability aspects of host-FPGA communication and depend on vendor-provided or custom device drivers PCIe communication.

III. Methods

A. Test case used: VirtIO Network device

In this work, we extend the implementation described in [14] to implement a VirtIO network device. This design uses the XDMA IP for PCIe connectivity. A VirtIO controller is placed between the XDMA IP and the user logic (as shown in Figure 2). The VirtIO controller implements the virtqueue functionality and controls the DMA engine of the XDMA IP. The DMA engine moves data between the host memory and the FPGA memory (BRAM or external DRAM). The VirtIO controller uses an interface that follows the same semantics as a virtqueue [13] to communicate with user logic. The user logic can interact with RX and TX queues provided by the VirtIO controller to send/receive data to/from the host.

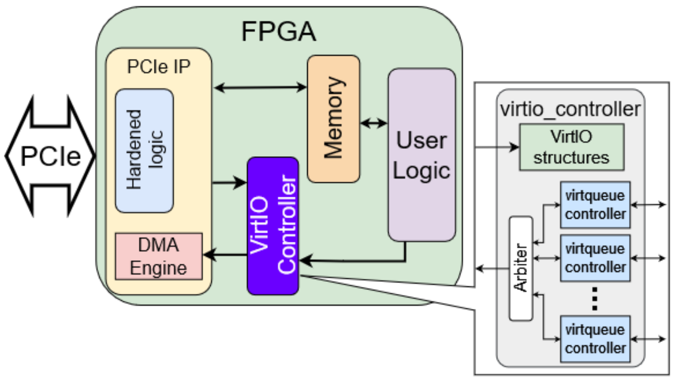

Fig. 2. VirtIO device architecture

Apart from data structures common to all VirtIO devices such as common configuration and notification, a device specific data structure is required to function as a particular device type. The device and the driver share information specific to the given device type using this data structure. For instance, the device specific data structure for a network device includes details such as the MAC address, Maximum Transmission Unit (MTU), and the types of hashes the device can calculate if the hash calculations for the incoming packets is offloaded to the device. The main modification to the design presented in [14] (to implement a VirtIO network device) is to implement the device-specific data structure. Depending on the features negotiated with the device driver, the device may also require a control queue apart from the RX and TX queues. However, no modifications are necessary to the VirtIO controller as the design already supports a variable number of queues.

When used as a network device, the FPGA receives Ethernet frames from the host. Depending on the features negotiated with the host during device initialization, the FPGA could either send out a received Ethernet frame as is or perform additional tasks on behalf of the host, e.g., a checksum calculation. Apart from offloading network functions, the FPGA can act as a SmartNIC onto which application-level tasks such as [30] can be offloaded. To enable application offloading to be done independently of the VirtIO drivers, we have (here) implemented an additional interface on the VirtIO controller that allows the user logic to request data transfers to/from host memory bypassing the VirtIO driver.

B. Experimental Setup

A Xilinx Artix-7 based Alinx AX7A200 PCIe development board (FPGA device number XC7A200TFBG484-2) is used as the target device. This board supports two PCIe Gen 2 lanes. The PCIe IP used on the FPGA is Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) [31]. The host machine is running the Fedora 37 operating system.

1) Testing VirtIO Drivers: We use the VirtIO network device implementation described in Section III-A to evaluate the performance of VirtIO device drivers when directly interacting with physical devices. This means that the host operating system recognizes and treats the FPGA as a NIC. The user space test application uses the C socket programming API to send packets to the FPGA. Entries are added to the operating system's routing table and ARP cache to facilitate routing packets from the test application to the FPGA.

2) Testing Vendor-Provided Device Drivers: An example design provided by Xilinx to demonstrate the XDMA IP core is used to test the reference device driver [12]. This design does not include any user logic; a BRAM is connected directly to an AXI memory-mapped interface of the PCIe IP, which enables the DMA engine to write/read to/from the BRAM. Minor modifications were made to change the width of the memory to match that used in the VirtIO design. This ensures the DMA engine can move data to and from FPGA memory at the same rate.

3) Metrics and Applications: The primary metric used to compare the different device drivers is the round-trip latency to move data to and from the FPGA. Since each FPGA design uses the same PCIe IP, and hence the same DMA engine, we expect the time taken by the DMA engine to move the same amount of data between the host and FPGA to be similar regardless of the device driver used. However, the time taken by the device driver to program the DMA engine and start the data movement can vary depending on design decisions made by the author of the device driver. We therefore infer that the device drivers themselves are largely responsible for any differences in data movement latency between the host and the FPGA. However, noise introduced by background processes executing on the host machine can also impact both the actual latency and any measurements made on the host side. Therefore, we have ensured that no other applications, except the test application, are running during the experiments. Each test consists of 50,000 packets for each payload size.

For time measurements, the test applications use the `clock_gettime()` function with the `CLOCK_MONOTONIC` option. For the system on which the tests were run, the timer resolution is 1ns. The PCIe IP and the VirtIO controller both include hardware performance counters to measure latency between different events on the FPGA. The FPGA designs used for testing are running at 125MHz. Therefore, the hardware performance counters provide a resolution of 8ns.

IV. Challenges, Workarounds, and Assumptions

Standalone latency measurements do not provide a complete picture of the performance of a real application implemented on the FPGA. There are several challenges in comparing the latencies of the two device drivers. Most of these arise from:

1) differences in design philosophies,
2) semantic differences in how the drivers are used, and
3) differences between the FPGA designs used for testing.

In the next three subsections, we first discuss these challenges and then describe the workarounds used, and assumptions made, to ensure fair and accurate comparisons.

A. Differences in device driver design

The XDMA driver is designed for a specific device and therefore includes many device-specific details such as the register space of the DMA engine and the descriptor format accepted by the particular DMA engine. It operates as a character device. At the most basic level, a user application can use the I/O system calls `read()`, and `write()` to move data between a buffer in the host memory and FPGA memory. The device driver then configures the DMA engine and initiates the DMA transfer.

The VirtIO drivers, however, are intended to target virtual devices, so their design does not take into account the existence of, or the necessity to program, a DMA engine. With VirtIO drivers, the back-end device, usually emulated by the host, is responsible for moving data between the buffers allocated by the front-end driver and itself. When VirtIO drivers are used to interact with physical devices, those devices become responsible for data movement to/from host memory. The finite state machine to control the DMA engine of the PCIe IP is part of the VirtIO controller (as shown in Figure 2).

The information required to program the DMA engine needs to be exchanged between the device driver and the device before initiating a DMA transfer. A major difference between the VirtIO and typical FPGA device drivers is when this information exchange takes place. When initiating a DMA transfer, the device driver creates one or more descriptors to provide the DMA engine with the source and destination addresses, buffer sizes, and any other control bits necessary. Depending on the capabilities of the DMA engine on the device, the driver can either provide a single descriptor at a time or an address for a descriptor table in host memory whence the DMA engine can fetch descriptors. Alternatively using the same descriptor table for all transactions and sharing the table address only at device initialization reduces overhead.

VirtIO drivers follow a different design philosophy in sharing information with the back-end devices. The driver shares the addresses of all the data structures necessary for virtqueue operation during device initialization. Therefore, to start a host-to-card (H2C) data transfer, only a notification using a single I/O write is needed at runtime. The device then accesses the data structures in host memory to determine how many new buffers were exposed by the driver and fetch buffer descriptors which it uses to perform data movement.

The differences are more pronounced with card-to-host (C2H) transfers. With the XDMA driver, the device interrupts the driver when it has data to be moved to the host memory. The user application uses a system call such as `poll()` to monitor the device file for interrupts and issues a `read()` call to initiate data movement. However, since a VirtIO device is aware of the location of all the necessary data structures in host memory, it can identify an available buffer and perform data movement before interrupting the driver.

These differences are inherent to the design of the two types of device drivers and we do not need to make adjustments to the latency measurements to account for them.

B. Differences in Device/application semantics

The second major difference between the VirtIO and vendor-provided device drivers is the semantics involved.VirtIO drivers come in different flavors to match different devices such as network devices, block devices, and many others [13].

The fundamentals of the VirtIO interface on the FPGA do not change based on the type of device implemented. Only the minimum number of queues and the device-specific configuration structure change across device types. Therefore, the modifications required to the FPGA design to support different device types are minimal. The main benefit of using semantics specific to different devices is the ability to leverage the host software stack for tasks that otherwise would have to be implemented in the user application.

For instance, assume that a user implements a SmartNIC using an FPGA. When using the VirtIO network device driver, the FPGA appears as a network interface card for the host OS. This means that a user application can use the host OS's network stack to send packets to the FPGA SmartNIC. In contrast, the vendor-provided XDMA device driver acts as a character device regardless of the application implemented on the FPGA. Therefore, to implement the SmartNIC, a user must either generate packets in the user application before using the device driver to move the generated packets to the FPGA, or write a new device driver that behaves like a network device.

This study uses a VirtIO network device to highlight the semantic differences described above. When using the VirtIO driver, the test program sends UDP packets to the FPGA using the C socket API. The user logic on the FPGA responds with a UDP packet of the same size. The test program measures the round-trip latency. Since the XDMA driver is a character device, the test program for the vendor-provided driver simply moves the same amount of data to the FPGA and back, and measures the round-trip time. Alternatively, it is possible to make the XDMA test program generate a packet before issuing a `write()` system call to move data to the FPGA. However, we have opted not to as the latencies recorded are similar despite the additional overheads associated with the VirtIO test case, e.g., generating packets and calculating checksums.

Hardware performance counters on the FPGA are used to measure the time taken by the hardware to perform the DMA operation once a notification is received. These times can be deducted from the latency measured by the test program to estimate the latency introduced by the software stack. For the VirtIO test, the time to generate the response packet is also deducted from the latency measurement since it is not relevant to the data movement latency. The buffer sizes for the VirtIO test program are set to ensure that the amount of data moved over the PCIe link to the FPGA is the same in both VirtIO and XDMA tests taking into account the protocol headers.

C. FPGA design

The FPGA designs used to test the two drivers differ in several ways. The difference that impacts the comparison the most is that the XDMA example design does not include user logic to generate interrupts for C2H data transfers. Therefore, the test application performs back-to-back H2C and C2H transfers without waiting for an interrupt from the device. This discounts the latency incurred by the XDMA driver to receive and handle two interrupts and underestimates the latency introduced by the XDMA driver in a real use case. While the vendor does provide another example design which includes logic to generate user interrupts for C2H transfers, this design generates the interrupts in response to an I/O write to the device. Since this introduces additional latency unnecessary for a real use case, this design is not considered. The final alternative is to implement a new design that receives data, monitors the DMA engine's status signals, and generates an interrupt when the H2C transfer is complete. This approach was not taken because that would increase the latency for the XDMA driver and the latency measurements for the two device drivers are comparable even with the favorable setup for the legacy driver.

V. Evaluation

This section presents and analyzes the results of the experiments described in Section III-B. Figure 3 summarizes the round trip latency distribution for different payloads when using VirtIO and vendor-provided XDMA device drivers. The payload varies between 64 Bytes and 1 KB. The payload sizes are selected such that the total latency is not dominated by the bus transactions and the effects of the drivers and the rest of the software stack are observable. The results show that the VirtIO driver provides performance comparable to the vendor-provided device driver despite the unfavorable experimental setup. Also, the VirtIO results show much lower variance.

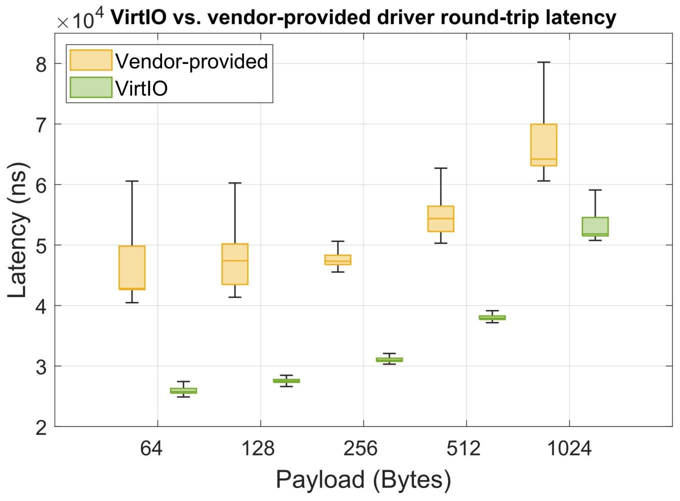

Fig. 3. Round-trip latency with VirtIO and vendor-provided device drivers.

Figure 4 presents a breakdown of the average round-trip latency for the VirtIO driver. The error bars represent the standard deviation. This shows that the time taken by the hardware to perform the DMA operations has minimal variance. Therefore, we can infer that the software stack is responsible for the majority of the variance in latency. It is also worth noting that the average latency for the software stack remains virtually constant throughout the range of payloads considered.

Figure 5 presents the same latency breakdown for the XDMA driver. An interesting distinction between the two latency breakdowns is that the time taken by the hardware is higher than the time for software with the VirtIO driver and vice versa with the XDMA driver. In the VirtIO use model, the back-end device performs data movement and does most of the work. In this scenario, the FPGA is the back-end VirtIO device. Therefore, it makes sense that the hardware performs more work when using the VirtIO driver. This difference could also explain the lower variance in the VirtIO latencies. As the variance in hardware latency is minimal, the setup that offloads more tasks to the hardware results in lower overall variance.

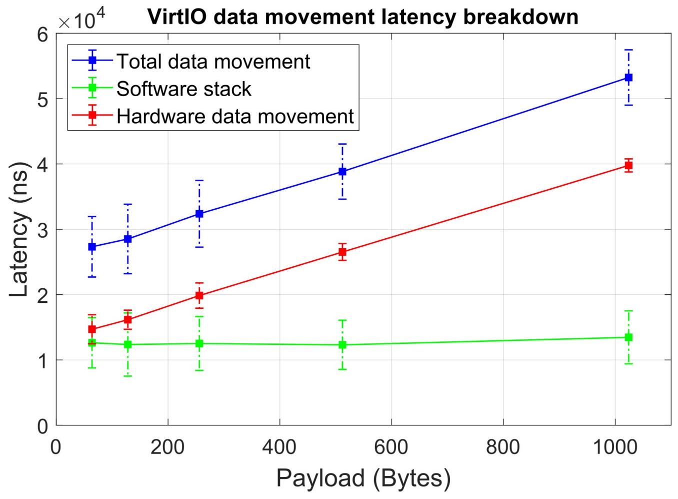

Fig. 4. Breakdown of data movement latency using the VirtIO driver.

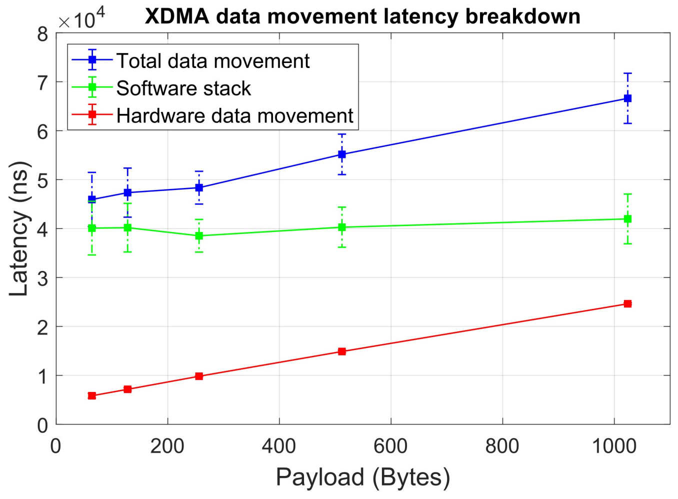

Fig. 5. Data movement latency breakdown with the vendor-provided driver.

Table I presents tail latencies for data movement with the two device drivers at different payloads. VirtIO shows lower tail latencies at 95 and 99 percentiles. However, there isn't a significant difference when we approach 99.9% tail latency.

TABLE I
TAIL LATENCIES FOR DATA MOVEMENT WITH VIRTIO AND XDMA.

Payload (Bytes)	95% (μs)		99% tail latency (μs)		99.9% (μs)	
	VirtIO	XDMA	VirtIO	XDMA	VirtIO	XDMA
64	35.1	51.3	44.8	70.1	66.5	85.8
128	33.6	51.4	48.1	60.0	88.4	88.2
256	39.6	51.5	53.8	57.5	75.1	70.6
512	44.1	59.1	57.4	64.5	82.1	87.5
1024	57.8	72.8	65.9	76.7	99.6	97.3

Our overall recommendation is as follows. For highly optimized applications with highly optimized software where low variance and tail latencies are critical, it is better to use a custom device driver. With such stringent requirements, the application is likely sufficiently important to be worth the additional cost of maintaining the driver. For all other everyday applications, however, VirtIO is preferred to vendor-provided reference drivers (including with possible minor changes).

VI. Conclusion

In this work, we have demonstrated that it is possible to replace vendor-provided or user-developed device drivers for FPGAs with generic in-kernel VirtIO drivers. The potential consequence is to significantly reduce the vast space of device-specific and custom FPGA device drivers. The performance analysis shows that in no case was the performance affected and in most cases it was marginally improved with reduced variance in data movement latency. However, the comparison was performed against a vendor-provided reference driver and a user could implement further optimized drivers based on it.

We now summarize the benefits of the proposed approach: 1) Using VirtIO drivers eliminates the requirement to write and maintain device drivers for FPGAs; 2) VirtIO drivers provide comparable performance to vendor-provided drivers; 3) VirtIO drivers make it easier to implement different types of devices and leverage the host OS's software stack for different tasks that otherwise would have to be implemented by the user application, probably with a loss of efficiency.

In conclusion, we recommend using custom drivers only for applications with strict performance requirements that far surpass the capabilities of vendor-provided reference drivers. For all other everyday applications, however, VirtIO is preferred to vendor-provided reference drivers to alleviate the overhead of maintaining device drivers.

We are currently in the process of performing the same experiments on different FPGA devices (different device families and from different vendors) and on different operating systems to demonstrate the portability of the proposed approach on both the device and host side.

This work is part of a bigger effort to enhance the usability of FPGAs through the automated generation of hardware operating systems using a specification of user requirements and component libraries as inputs. The generator is called the "Dynamic Infrastructure Services Layer" (DISL), which generates a layer of hardware implementing services such as memory, I/O, and host interfaces to be used by a user application implemented on the FPGA.

Acknowledgments

This work was supported, in part, by the NSF through awards CCF-1919130 and CCF 2151021; and by Red Hat through awards 2023-01-RH13 and 2024-01-RH08.

References

[1] A.M. Caulfield, et al., "A cloud-scale acceleration architecture," in *MICRO*, 2016.
[2] Q. Xiong, C. Yang, R. Patel, T. Geng, A. Skjellum, and M. Herbordt, "GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs," in *FCCM*, 2019, pp. 258–266.
[3] J. Lant, J. Navaridas, M. Lujan, and J. Goodacre, "Toward FPGA-Based HPC: Advancing Interconnect Technology," *IEEE Micro*, vol. 40, no. 1, pp. 25–34, 2020.
[4] V. Krishnan, O. Serres, and M. Blocksome, "Configurable Network Protocol Accelerator (COPA)," *IEEE Micro*, vol. 41, no. 1, 2021.
[5] C. Bobda, et al., "The Future of FPGA Acceleration in Datacenters and the Cloud," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 3, pp. 1–42, 2022, doi: 10.1145/3506713.
[6] P. Haghi, et al., "Reconfigurable switches for high performance and flexible MPI collectives," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 2, 2022.
[7] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt, "A Framework for Neural Network Inference on FPGA-Centric SmartNICs," in *FPL*, 2022.
[8] C. Wu, T. Geng, A. Guo, S. Bandara, P. Haghi, C. Liu, A. Li, and M. Herbordt, "FASDA: An FPGA-Aided, Scalable and Distributed Accelerator for Range-Limited Molecular Dynamics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607100
[9] PCI SIG Org., "PCI Express Base Specification Revision 3.0," Nov 2010. [Online]. Available: https://pcisig.com/specifications
[10] "Active kernel releases." [Online]. Available: https://www.kernel.org/category/releases.html
[11] "Red Hat Enterprise Linux Release Dates," Nov 2023. [Online]. Available: https://access.redhat.com/articles/3078
[12] Xilinx, "Xilinx DMA IP Reference drivers." [Online]. Available: https://github.com/Xilinx/dma_ip_drivers
[13] M. S. Tsirkin and C. Huck, "Virtual I/O device (VIRTIO) version 1.2," May 2022. [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html
[14] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt, "Enabling VirtIO Driver Support on FPGAs," in *2022 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2022, pp. 1–8.
[15] Xilinx Inc., "Xilinx Run Time for FPGA," 2022. [Online]. Available: https://github.com/Xilinx/XRT
[16] E. Luebbers, S. Liu, and M. Chu, "Simplify Software Integration for FPGA Accelerators with OPAE." [Online]. Available: https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf
[17] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, "Deploying multi-tenant FPGAs within Linux-based cloud infrastructure," *TRETS*, vol. 15, no. 2, pp. 1–31, 2021.
[18] J. M. Mbongue, F. Hategekimana, D. T. Kwadjo, and C. Bobda, "FPGA Virtualization in Cloud-based Infrastructures over Virtio," in *ICCD*, 2018, pp. 242–245.
[19] "P-Tile Avalon Streaming Intel FPGA IP for PCI Express User Guide." [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683059/23-4-9-1-0/about-the-p-tile-fpga-ips-for-pci-express.html
[20] Silicom, "Silicom C5010X data center NIC." [Online]. Available: https://www.silicom-usa.com/wp-content/uploads/2021/12/C5010X-Data-Center-FPGA-IPU-NIC.pdf
[21] RSPwFPGAs, "Virtio-FPGA-Bridge: Virtio front-end and back-end bridge, implemented with FPGA." [Online]. Available: https://github.com/RSPwFPGAs/virtio-fpga-bridge

[22] D. Korolija, T. Roscoe, and G. Alonso, “Do OS abstractions make sense on FPGAs?” in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 991–1010.

[23] C. Heinz, J. Hofmann, J. Korinth, L. Sommer, L. Weber, and A. Koch, “The TaPaSCo Open-Source Toolflow: for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems,” *Journal of Signal Processing Systems*, vol. 93, pp. 545–563, 2021.

[24] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, “FOS: A modular FPGA operating system for dynamic workloads,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 4, pp. 1–28, 2020.

[25] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, “ReconOS: An operating system approach for reconfigurable computing,” *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2013.

[26] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, “Enabling a uniform programming model across the software/hardware boundary,” in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 89–98.

[27] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, “The Feniks FPGA operating system for cloud computing,” in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, pp. 1–7.

[28] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, “A Hypervisor for Shared-Memory FPGA Platforms,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[29] K. Fleming and M. Adler, “The LEAP FPGA operating system,” *FPGAs for software programmers*, pp. 245–258, 2016.

[30] Z. Tahir, A. Sanaullah, S. Bandara, U. Drepper, and M. Herbordt, “Multi-core Multi-rule VeBPF Firewall for Secure FPGA IoT Device Deployments,” in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024.

[31] Xilinx, “DMA/Bridge Subsystem for PCI Express v4.1,” Jun 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/pg195-pcie-dma