A Full-System VM-HDL Co-Simulation Framework for Servers with PCIe-Connected FPGAs

Shenghsun Cho, Mrunal Patel, Han Chen, Michael Ferdman, Peter Milder
Stony Brook University

ABSTRACT

The need for high-performance and low-power acceleration technologies in servers is driving the adoption of PCIe-connected FPGAs in datacenter environments. However, the co-development of the application software, driver, and hardware HDL for server FPGA platforms remains one of the fundamental challenges standing in the way of wide-scale adoption. The FPGA accelerator development process is plagued by a lack of comprehensive full-system simulation tools, unacceptably slow debug iteration times, and limited visibility into the software and hardware at the time of failure.

In this work, we develop a framework that pairs a virtual machine and an HDL simulator to enable full-system co-simulation of a server system with a PCIe-connected FPGA. Our framework enables rapid development and debugging of unmodified application software, operating system, device drivers, and hardware design.

Once debugged, neither the software nor the hardware requires any changes before being deployed in a production environment. In our case studies, we find that the co-simulation framework greatly improves debug iteration time while providing invaluable visibility into both the software and hardware components.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; **Functional verification**; *Hardware-software codesign*; • **Computer systems organization →** *Heterogeneous (hybrid) systems*;

ACM Reference Format:
Shenghsun Cho, Mrunal Patel, Han Chen, Michael Ferdman, Peter Milder. 2018. A Full-System VM-HDL Co-Simulation Framework for Servers with PCIe-Connected FPGAs. In *FPGA'18: The 2018 ACM / SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3174243.3174269

1 INTRODUCTION

FPGAs are gaining popularity as an accelerator technology to offload complex computation and data flows. The combination of programmability, a high degree of parallelism, and low power consumption make FPGAs suitable for environments with rapidly changing workloads and strict power consumption limits, such as data centers. To put FPGAs into existing systems, PCIe has become the most common connection choice, due to its wide availability in server systems. Today, the majority of FPGAs in data centers are communicating with the host system through PCIe [2, 12].

Unfortunately, developing applications for PCIe-connected FPGAs is an extremely slow and painful process. It is challenging to develop and debug the host software and the FPGA hardware designs at the same time. Moreover, the hardware designs running on the FPGAs provide little to no visibility, and even small changes to the hardware require hours to go through FPGA synthesis and place-and-route. The development process becomes even more difficult when operating system and device driver changes are required. Changes to any part of the system (the OS kernel, the loadable kernel modules, the application software or hardware) can frequently hang the system without providing enough information for debugging, forcing a tedious reboot. The combination of these problems results in long debug iterations and a slow development process, especially in comparison to the quick iteration of the software development process familiar to application developers.

The traditional way to test and debug hardware designs without running on a hardware FPGA platform is by writing simulation testbenches, either using Hardware Description Languages (HDLs) or Hardware Verification Languages (HVLs), sometimes combined with Bus Functional Models (BFMs) provided by the FPGA vendors. However, this approach prevents the hardware from being tested together with the software and operating system. Moreover, writing testbenches with high coverage is an extremely time-consuming and error-prone process. While some vendors provide hardware-software co-simulation environments [3, 5, 7, 18], these environments skip the operating system and driver code and provide a limited environment for development, typically restricting the use of the co-simulation to simple designs while still requiring considerable development and debugging effort for porting the designs from the co-simulation framework to the hardware FPGA platform. There have been several efforts to provide frameworks to connect instruction-set simulators to HDL simulators to perform full-system simulation. However, full-system simulation of datacenter servers is itself an open research challenge and the speed of full-system simulation of the servers alone limits their use for software development [17]. As such, existing full-system simulation-based frameworks target system-on-chips—typically, ASICs with ARM cores—as tools for early stage design exploration, software development, and functional verification, rather than the cycle-accurate simulations required to verify or debug FPGA-accelerated servers.

We observe that, although there are no available full-system simulation environments of servers with PCIe-connected FPGAs, we can extend existing mature and robust tools to build a co-simulation framework for rapid development and debugging of such systems. In particular, modern datacenters employ virtual machines (VMs) in production environments to provide security and isolation. Because of this, virtual machines are effective for high-performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA'18, February 25–27, 2018, Monterey, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174269

emulation of full server systems, including CPUs, disks, memory, and network interfaces. This makes VMs a natural fit to emulate the server system in an FPGA development environment. Moreover, FPGA vendors provide sophisticated HDL simulation software, allowing cycle-accurate simulation of the designs as they would run on the target hardware FPGA platform. Although virtual machines and HDL simulators separately provide effective software and hardware development environments, the key missing enabler is a link between a VM's virtual PCIe and NIC devices and the PCIe and network blocks in an HDL simulation platform. This link should be transparent both to the operating system and software running inside the VM, and to the hardware design in the HDL simulator.

In this work, we developed a co-simulation framework by providing PCIe and NIC communication channels between a VM and an HDL simulator. On the VM side, we created a software pseudo device to represent the FPGA and proxy all device interactions to the HDL simulator. The operating system and software running inside the VM see the same PCIe-attached device as if they were running in a hardware system with an FPGA plugged in.

On the HDL side, we developed a PCIe simulation bridge to communicate with the VM and proxy all hardware events to the software pseudo device in the VM. The PCIe simulation bridge is pin-compatible with the PCIe block of the hardware FPGA platform. Similarly, we developed a pin-compatible NIC simulation bridge to allow the FPGA simulation to exchange Ethernet frames with a host network. The FPGA design observes the same interfaces toward PCIe and network; thus it requires no modification or porting to work with the co-simulation framework. To the FPGA development tools, the PCIe and NIC simulation bridges appear as regular hardware blocks and have no impact on the simulation flow.

To demonstrate our VM-HDL co-simulation framework, we prototyped two server systems: a sorting accelerator and a network card, both using a PCIe-connected FPGA. Our experience indicates that the co-simulation framework significantly reduces the debug iteration time and enables rapid design exploration and debugging that was not previously possible with the available vendor tools. Moreover, our framework provides invaluable visibility into both the hardware design and the operating system, making it easier and faster to identify problems while developing and debugging. The framework allows single-stepping the host kernel software instruction by instruction, and examining variable contents and interactions with the hardware, while simultaneously recording and visualizing all signal waveforms in the hardware design.

The rest of this paper is organized as follows: Section 2 provides an in-depth discussion of co-simulation requirements and the motivation behind our approach. Section 3 describes our co-simulation framework. Section 4 describes the framework implementation. Section 5 presents our case studies and evaluation. Lastly, Section 6 discusses related work and Section 7 gives concluding thoughts.

2 MOTIVATION AND APPROACH

The continually rising demand for data processing in datacenters fuels the need for efficient accelerators. However, to achieve widespread adoption, accelerator technologies must not only achieve high performance and efficiency, but they must also be convenient for application developers. Engineers and their managers have specific expectations of what constitutes a practical development environment and reasonable debug iteration time, stemming from their experience in software development for CPUs and GPUs. Unfortunately, even ignoring the fact that FPGA development requires a unique and expert set of skills beyond those of a typical software engineer, the development of the hardware and software needed for FPGA acceleration in datacenters presents fundamental challenges in terms of full-system test, debug iteration time, and visibility into the design internals during development.

2.1 Challenges of Development and Debugging

Full-System Test. A major challenge with existing FPGA development environments is the inability to simulate full systems including the OS, device drivers, application software, network, and the accelerator logic. As a result, accelerators are developed in isolation from the software. In traditional FPGA design, developers rely on simulation and carefully-crafted hardware testbenches. However, the complexity of server systems and the expected fluidity of rapid prototyping during the early stages of system development make testbench-driven design impractical for datacenter applications.

Existing SoC co-simulation environments address this problem by simulating embedded CPUs with the entire software stack. However, performing per-instruction CPU emulation in hardware simulation cannot be used for production server software, as simulation is many orders of magnitude less than "interactive" performance expected by developers [17]; even if a simulation platform could be configured to emulate a typical multi-core x86-64 server system, it would take days to just boot the server in such an environment.

Seeing these problems, vendors targeting datacenter accelerators have adopted development platforms that connect HDL simulators with user-level libraries that wrap accelerator functionality. However, the interfaces used for these libraries are non-standard and brittle, developed by small teams for very specific use cases and a very limited user base. Once an initial prototype is developed in such an environment, a significant porting and debugging effort must still take place to move the design to the hardware FPGA platform and integrate it with the production software stack.

Moreover, FPGA accelerators in datacenters may have network connectivity beyond the host server. To support network connectivity, FPGA vendors provide IP blocks that interconnect on-chip data streams with off-chip high-speed interfaces, including all functionality (e.g., MAC, PCS, etc.). Developing and testing network functionality with hand-written testbenches is impractical, because the testbench would need to supply many carefully-crafted network packets whose contents depend on the responses received from the hardware design. As a result, these systems are currently developed and debugged directly on hardware FPGA platforms.

Debug Iteration Time. After a design is implemented on the hardware FPGA platform and integrated with the software stack, the next challenges are debugging and rapidly iterating over design changes. A fundamental challenge is caused by the difficulty of synthesizing, placing, and routing a design for large modern FPGAs. For example, despite large-scale investments in FPGA infrastructure from Microsoft [4], Intel [6], and Amazon [2], the environments used to develop accelerators in these frameworks are not amenable to rapid iteration. Each change to the FPGA design requires hours

of waiting before the new design can be tested. While such times are acceptable in an ASIC or embedded appliance development environment, wide-scale adoption of FPGAs for datacenter applications is severely stunted by such slow development practices.

Considering these challenges alongside the difficulties associated with full-system test, FPGA designers face a difficult dilemma: either attempt to debug in simulation (using testbenches and isolating the hardware from the software stack), or work with the production hardware and software, sacrificing the ability to iterate quickly.

Visibility. In addition to the needs of fast iteration and running full-system tests throughout the development process, rapid accelerator development requires visibility into both the server software and the accelerator hardware as expected for traditional software development. However, existing platforms offer limited introspection for system software and hardware. For example, running tests on hardware FPGA platforms limits visibility if the system freezes, which leads to debugging with printf() and frequent rebooting.[1] On the hardware side, tools such as SignalTap [1] and Integrated Logic Analyzer [20] permit the insertion of probes on a select subset of signals in the hardware design. However, the number of signals that can be probed is limited, the amount of time for which signals can be recorded is limited, and changing the set of signals monitored (or in some cases, the trigger events) requires place and route of the design. Developers must wait for hours before re-running an experiment whenever an additional signal needs to be collected.

2.2 Our Approach

In this work, we set out to create a co-simulation framework that allows for full-system test, rapid debug iteration, and complete visibility into the system under development. Moreover, our goal was to construct a flexible platform based on robust, mature, and production-ready components, so that it can be quickly adopted by a wide range of projects without discouraging accelerator developers due to platform limitations, brittleness, and poor performance. To this end, we developed a co-simulation framework that connects a virtual machine (running production server software) with an HDL simulation (running unmodified accelerator hardware designs). The resulting system addresses the challenges described above.

Enabling Full-System Test. Our co-simulation framework enables development and debugging to take place in a full-system environment without any software modifications to the operating system, device drivers, and application software and without any modifications to the accelerator hardware design. Precisely the same software and HDL code are used in the development environment as the production environment. All software components and any part of the accelerator hardware can be modified and debugged in co-simulation. This allows all components of the design to be seamlessly moved between the production environment and the co-simulation environment without modification.

Our system employs a virtual machine (VM) that mimics the production server environment. The VM can use CPU virtualization features, which allow it to execute nearly as fast as bare-metal hardware. In fact, in some cases, such as during the system reboot process, the VM runs faster than bare metal due to the host's disk cache, which can service requests faster than a physical disk. This setup provides a functionally-correct fast and convenient development environment. Additionally, to help developers debug more obscure timing-sensitive problems, our system supports transitioning in and out of "lock-step" mode, where the VM's execution speed is paired with the HDL simulation's clock.

In addition to the PCIe connection to the host, FPGA platforms often include connections to Ethernet networks. For example, the latest Microsoft Catapult FPGAs use a bump-in-the-wire arrangement [4] where network traffic intended for the host passes through the FPGA fabric on its way to the host's built-in NIC. Other prominent FPGA platforms, such as the NetFPGA SUME [23] and the Xilinx VCU118 [19], feature multiple network ports that can be used to configure these platforms as a "Smart NIC" for the host.

To support debugging systems with external network connections, our co-simulation framework includes a network interface simulation bridge, providing network connectivity to the FPGA. Virtual machines have extensive support for network connectivity through software-defined networking (SDN) components. We leverage this infrastructure to link the HDL simulator into the VM network, enabling the developer to use standard SDN tools to bridge the accelerator hardware running in simulation into test networks (including real LANs or virtual LANs), and even provide direct connectivity between the simulated hardware and the public Internet. This feature is invaluable for debugging network-connected FPGAs, as it allows the developer to make the FPGA participate in real bi-directional network traffic during development and debugging.

Reducing Debug Iteration Time. The use of standard HDL simulation tools for the accelerator design allows hardware designers to use a familiar simulation workflow. Importantly, modifications to the accelerator HDL sources can be done quickly, avoiding the synthesis and place-and-route process which would be necessary when targeting a hardware FPGA platform. Changes to the hardware design require only a quick rebuild or restart of the HDL simulation infrastructure, which ranges between seconds to at most several minutes, depending on the design complexity.

The software and hardware simulation components can be restarted independently. During the debugging process, developers frequently face the need to restart application software and unload and reload kernel device drivers, which can be done freely in our co-simulation framework. In addition to these, sometimes it is necessary to reboot the system or reset the hardware (for example, when the operating system source code or the hardware design are modified). By connecting the virtual machine and HDL simulator using a fault-tolerant high-level message queue implementation, either side of the co-simulation can be independently restarted and the sides will automatically reconnect and continue communicating as though they were never disconnected or modified.

Providing Full Visibility. The co-simulation framework offers complete visibility into both software and hardware. In software, the VM environment supports a remote gdb debugger interface that permits temporarily freezing the VM and stepping through the source code of the software running inside, line by line or instruction by instruction, and observing variable contents in memory. In hardware, HDL simulators offer the capability of tracing all signal waveforms from the beginning of simulation without requiring simulation restarts or prior selection of specific signals to monitor.

[1]The situation is exacerbated further for bump-in-the-wire systems [4], where an error in the FPGA logic can lead to complete loss of connectivity to the server.

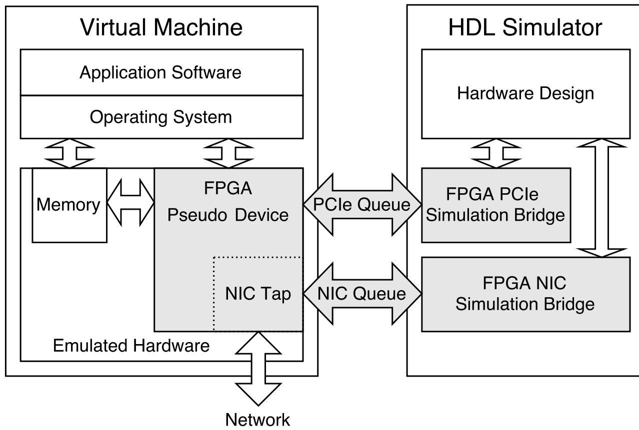

Figure 1: Our VM-HDL co-simulation framework (new components we developed are shaded gray)

3 VM-HDL CO-SIMULATION FRAMEWORK

We built our VM-HDL co-simulation framework by coupling several mature technologies. A high-level architecture of the framework in shown Figure 1. On the host side (left), a virtual machine is used in place of the server. On the FPGA side (right), the hardware design runs in a commercial HDL simulator. The key framework components are the links between the hypervisor[2] and the HDL simulator, comprising the *PCIe link* between the server and the FPGA and the *NIC link* between the FPGA and the Ethernet network. Critically, these links provide exactly the same interfaces and functionality as their hardware FPGA platform counterparts, allowing seamless back-and-forth transitions between the co-simulation framework and deployment on production hardware. All other parts of the system, including the FPGA accelerator design, operating system, device drivers, and application software run in co-simulation and on the production hardware without any modifications.

3.1 FPGA Pseudo Device

On the VM side, we created an FPGA pseudo device module for the hypervisor. From the perspective of the hypervisor, this module emulates a hardware platform's FPGA and its interfaces (PCIe and NIC). To the guest operating system running inside the virtual machine, the pseudo device appears exactly like a PCIe-connected FPGA in the target platform. The device exposes the same number and size of the Base Address Register (BAR) regions and Message Signaled Interrupt (MSI) capabilities. The hypervisor interacts with the pseudo device using high-level abstractions such as Memory-Mapped I/O (MMIO), Direct Memory Access (DMA), and interrupts. We maintain this high level of abstraction when sending these operations over the communication links, thereby avoiding low-level protocol details.

A typical NIC module for a hypervisor has two interfaces. One exposes the NIC to the guest VM as a PCIe device, and the second exposes the NIC to the host system as a *tap* network interface. To support debugging FPGA designs with network interfaces, our pseudo device creates a host tap network interface and relays Ethernet frames between this interface and the NIC link connected to the HDL simulator. This enables the developer to use standard software defined networking tools such as bridges and virtual switches to connect the exposed tap interface to virtual or physical networks.

[2]We use the term *hypervisor* to refer to the software application used to emulate the virtual hardware and launch a virtual machine.

3.2 FPGA PCIe Simulation Bridge

On the HDL side, we developed an FPGA PCIe simulation bridge to replace the PCIe bridge IP in the FPGA platform. The PCIe simulation bridge is pin-compatible with the PCIe bridge IP provided by the FPGA vendor, exposing exactly the same interface and functionality to the FPGA hardware design. Because of this, all interactions between the FPGA hardware and the host are identical in the co-simulation and production environments. Because the interface is unchanged, the FPGA hardware design is entirely unaware of the fact that it is operating in simulation. No modifications are required to run hardware designs in co-simulation, and the design can glide freely between the production environment and co-simulation.

The PCIe bridge offers three functions: forwarding the host's MMIO requests to the FPGA interconnect, forwarding the FPGA interconnect's DMA requests to the host, and raising interrupts on the host in response to the bridge interrupt pins. Notably, the PCIe bridge operates at two different levels of abstraction on its two sides. When communicating to the hypervisor, the bridge maintains the same high level of abstraction as the FPGA pseudo device. The data transfers use high-level operations rather than low-level PCIe transactions. However, on the side of the hardware design, the bridge faithfully emulates the hardware FPGA platform and uses the same low-level cycle-accurate protocol models to receive messages from (and deliver messages to) the FPGA interconnect.

3.3 FPGA NIC Simulation Bridge

Support for the bump-in-the-wire and NIC scenarios in a full-system setup is facilitated by an FPGA NIC Simulation Bridge. Just as the PCIe bridge shuttles PCIe operations between the hypervisor and HDL simulation, the NIC bridge is responsible for shuttling Ethernet frames between the network components of the hardware design in simulation and the hypervisor.

The NIC bridge is pin-compatible with the vendor provided IP, which handles all protocol details and expects for the off-chip interfaces to connect to an Ethernet network. Our co-simulation NIC bridge interacts with the rx and tx streams in the same way as the vendor-provided IP. However, rather than interacting with a physical network, the received and transmitted frames are communicated as high-level operations to the hypervisor NIC tap functions.

Although our current prototype assumes Ethernet networks, we note that this approach is not limited to a specific protocol and is not restricted to communication only with other VMs and servers. For example, a NIC bridge can be used to connect multi-FPGA systems in the co-simulation framework. Multiple concurrently-running HDL simulators, each with its own NIC bridge, can be used for co-simulation of a system like the Amazon F1.16xlarge [2] instance, where each FPGA connects to the server host via PCIe and to the other FPGAs via a 400Gbps bi-directional ring.

3.4 VM-HDL Link Queues

The PCIe link and NIC link provide communication between the hypervisor and HDL simulator. These links can be implemented using domain or network socket APIs. However, rather than directly relying on a low-level stream protocol, we construct the links using a high-level message queuing library. Messages are guaranteed to be reliably delivered to the destination process in their entirety and communication is non-blocking, allowing the sending process to continue running after enqueuing a message.

Beyond simplifying the implementation, the queue-based approach offers functional benefits. The queue abstraction allows independently restarting the VM or the HDL simulator (e.g., after making changes to the hardware design). The system automatically re-establishes the connections and continues exchanging high-level messages after restart. The queue interface also allows to independently pause and resume execution of the VM or HDL simulator (for example, to examine their internal state), without incurring timeouts and aborts from the side that is not paused.

3.5 Untimed and Lock-Step Modes

Our co-simulation framework enables an optional *lock-step* mode, which forces the VM and the HDL simulation to have a consistent view of time. This option can be invaluable in resolving timing bugs or verifying complex time-sensitive interactions between the server software and hardware design. This mode contrasts with the standard *untimed* mode, where the VM operates on real-world time and the HDL simulator advances time as fast as it can.

Lock-step mode was easily added to the co-simulator framework because both the hypervisor and the HDL simulator have mechanisms to control the advance of time. On the software side, we can place a hypervisor into a mode where it acts similar to an instruction-set simulator, with a mechanism to advance time one instruction at a time. On the hardware side, we can easily control when the clock signal advances in the HDL simulation environment. By adding an extra link between them, we can synchronize the passage of time, allowing them to proceed in lock-step according to a user-provided clock ratio (e.g., eight server cycles to one FPGA cycle to simulate a 2GHz server with a 250MHz FPGA).

The downside to this approach is that using a hypervisor in this mode imposes a very high performance overhead. Our system aims to minimize the effect of this by allowing the developer to switch between untimed and lock-step mode dynamically, as needed.

3.6 Debugging and Development Interfaces

Our co-simulation framework enables extensive debugging and development capabilities that take the process of working with PCIe-connected FPGAs a step closer to the simplicity and ease-of-use of traditional software-development environments.

For software debugging, the hypervisor can act as a remote target for command-line and graphical debuggers. Developers can use familiar tools such as *gdb* to connect to the hypervisor to examine the contents of memory and registers inside the virtual machine and for single-step execution. Remote target support allows debugging not only the application software running within the VM, but also to debug the device drivers and operating system, including interrupt handlers, providing complete access to memory and registers and supporting single-stepping at both the C statement and the assembly instruction granularity. Moreover, the interface allows the developer to modify memory and register contents on the live system to experiment with various scenarios and on-the-fly fixes.

For hardware debugging, developers can use the HDL simulator to record all hardware signals during the entire simulation. As a result, the co-simulation framework provides greater visibility than using an in-hardware virtual logic analyzer, which limits the number of probed signals and requires place-and-route to add probes. Our approach not only provides full visibility into all signals in the system at the current time, but also allows examining the HDL state at any point in the past, enabling the developer to quickly trace back and identify the origin of a bug, regardless of how far in the past it occurred. Similar to the software debug interface, the HDL simulator also supports single-step operation, examining all register contents, and forcing signal and memory values.

In addition to enabling the classic approaches above, our co-simulation infrastructure offers a new hybrid mode of development and debugging, combining the practices and expectations of developing on a hardware FPGA platform with the debugging capabilities of simulation. Specifically, software and hardware developers find it natural to edit, compile, run, and debug code directly on the target platform. Because of the high (near bare metal) performance of the VM in co-simulation, including an additional network interface in the VM allows it to be used by the developers just like a hardware FPGA platform. The VM can fulfill all expectations of a traditional development environment by including all editors and build tools needed to work with code, mounting remote NFS or SMB filesystems, and allowing interaction with remote code repositories. When the co-simulation framework is deployed in a datacenter, developers can use ssh or remote desktop to log into the co-simulation VM and use it exactly as if logging into a hardware FPGA platform in a production cloud environment. However, while the platform behaves like a server system with a PCIe-connected FPGA, it comes with the added benefits of full visibility into the hardware waveforms and the ability to make changes to the hardware design and see them immediately reflected on the live system.

4 FRAMEWORK IMPLEMENTATION

In this section, we describe our design decisions and provide the key implementation details of our co-simulation framework.

4.1 Hypervisor

To emulate the server system, we use QEMU, an open-source hypervisor that is widely deployed in production environments. QEMU includes many features that make it particularly well-suited for this work: it has robust emulation models of server system components, it provides a rich API for developing new device models such as the FPGA Pseudo Device, it offers bare-metal speeds using hardware-accelerated virtualization via KVM [8], it includes an instruction-set simulator-like mode to enable single-step execution, and it supports a standard remote target debugger interface.

FPGA Pseudo Device. To allow the VM to interact with the simulated FPGA device, we developed a QEMU virtual device to serve as a proxy for the FPGA, supporting PCIe transactions and

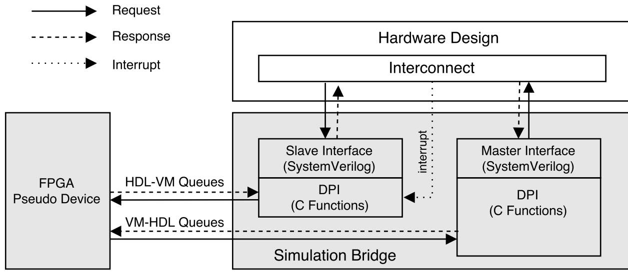

Figure 2: FPGA PCIe simulation bridge

network communication. We based our implementation on a QEMU reference design of a NIC. The QEMU API supports customization of all PCIe device parameters, enabling our implementation to exactly mimic the target PCIe-connected FPGA system. The pseudo device identifies itself using the vendor and device ID of the hardware FPGA platform, and matches the Base Address Register (BAR) address widths, sizes, pre-fetch capabilities, and the number of Message Signaled Interrupt (MSI) interrupts. For NIC devices, the API also includes functions to declare network interfaces, read and configure their MAC addresses, and send out Ethernet frames.

To enable the device to respond to events, QEMU provides an interface for the device model to register callback routines. Our implementation registers callbacks for handling the VM's MMIO reads and writes on the FPGA BAR regions and a callback for receiving Ethernet frames from the network interfaces. Additionally, QEMU supports registering file descriptors with its event loop and triggering custom callbacks when there is activity on those file descriptors. We leverage this functionality to enable efficient receipt of messages from the HDL simulator by registering callbacks on the network sockets underlying our PCIe and NIC message queues.

Network Interfaces. QEMU network devices create a *tap* virtual network interface on the host. Frames sent by the device model are transmitted by QEMU on the tap interface. Frames received by the tap interface trigger a QEMU callback, which forwards frames to the HDL simulator. The tap interface is the standard mechanism for virtual machine network connectivity. It can be bridged into the physical network of the simulation host via software-defined networking components (e.g., a Linux bridge), in a manner identical to a production setup in a cloud environment. From the perspective of all other devices connected to the same network, the simulated hardware is indistinguishable from a plugged-in physical device. Alternatively, the tap interface can be bridged into a private virtual network, together with other VMs running on the same host. These VMs can use a full-system software stack to act as a traffic generator and to expose the hardware design to a variety of test loads.

When the tap interface is bridged into a physical Ethernet network, the hardware design in the co-simulation framework is exposed to all traffic from this network segment. Such networks routinely observe a significant amount of background chatter, such as ARP requests and other broadcast protocols, which serves as an excellent way to expose the hardware design to a diversity of real packet contents and timing scenarios in addition to the test traffic.

4.2 Hypervisor-Simulator Message Queues

Rather than directly relying on a stateful connection-oriented bi-directional stream protocol, we link the QEMU and the HDL simulator using pairs of unidirectional high-level message queues constructed with the ZeroMQ (ZMQ) messaging library [22]. ZMQ is a high-level message library that wraps the low-level details of inter-process communication. The library provides reliable message delivery, which is particularly helpful when one side of the co-simulation framework slows down, crashes, or simply needs to be restarted, allowing the other side to continue without interruption. The loose coupling of the processes enables high performance operation and makes the system very robust, freeing our implementations of the pseudo device and simulation bridges from the responsibility of handling incomplete messages and flow control.

Each communication link comprises a pair of unidirectional channels. There are two communication links for performing PCIe operations between QEMU and HDL simulation: one for QEMU to HDL simulator messages and another for HDL simulator to QEMU messages. The NIC bridge uses two unidirectional channels, one for transmitting outgoing frames and one for receiving incoming frames. Each message sent over a channel contains a structure comprising the operation type and attributes, such as the address offset, data, data length, BAR number, etc. The structure and content of the messages can be easily modified or extended, as the details of reliably delivering the messages is handled by ZMQ.

4.3 Co-Simulation Bridge IPs

The co-simulation bridge IPs serve as interfaces between the hardware design and the hypervisor, linking the HDL simulation with QEMU (via ZMQ channels). The bridges are built using SystemVerilog's direct programming interface (DPI), which allows interactivity between the HDL simulation and external software. The bridges are pin-compatible with Xilinx-provided IPs for PCIe and Ethernet controllers. The bridges are parameterized to allow them to be easily configured to match different hardware FPGA platforms (e.g., PCIe interfaces with differing numbers of lanes).

FPGA PCIe Simulation Bridge. Figure 2 shows a block diagram of the FPGA PCIe simulation bridge. Like the Xilinx PCIe bridge IP, the simulation bridge has AXI slave and master interfaces and an MSI interrupt input. The master interface facilitates MMIO requests from the host to the hardware design and the slave interface supports memory requests from the hardware to the host.

In the simulation bridge, each interface's functionality is split between SystemVerilog code (which drives the hardware-facing AXI interfaces), and C functions (which interact with the ZMQ channels to communicate with QEMU); SystemVerilog DPI is used to link the two. The slave interface code is activated by the (simulated) clock. On each positive clock edge, the interface module checks for a new request from the hardware design via its AXI port. When an AXI read or write is detected, the interface calls a C function, which translates the request to a high-level message for the FPGA pseudo device and places the message into the HDL-VM ZMQ channel.

To handle responses coming back from QEMU, the interface calls a C function to poll the ZMQ response channel. The polling function is invoked on the positive clock edge of each simulated cycle. When it detects a response on the channel, the C function triggers a state

machine within the interface module, which feeds the response data into the corresponding AXI port. Lastly, the slave interface has an MSI interrupt input port. Whenever the SystemVerilog code detects that the interrupt is raised, it calls a C function to write the interrupt request message into the HDL-VM channel.

The master interface works similarly. The interface uses a clock-edge activated SystemVerilog block to call a C function which polls the VM-HDL channel for MMIO requests from QEMU; when one is detected, the C function triggers a state machine that feeds the request data into the AXI port. Responses (which arrive from the hardware via the AXI port) are detected on positive clock edges and are sent (using a C function) to the VM-HDL response channel.

FPGA NIC Simulation Bridge. The NIC bridge has a similar structure to the PCIe bridge, with two notable differences. First, the NIC bridge only needs one ZMQ channel pair to handle incoming and outgoing frames. Second, the NIC bridge uses unidirectional AXI Stream interfaces to interact with the hardware design (rather than connecting to an AXI interconnect). The NIC bridge indicates frame boundaries by setting the Start of Frame (SOF) and End of Frame (EOF) symbols on the rx stream and uses the EOF symbol on the tx stream as an indication that a complete frame has been received from the hardware and should now be sent to QEMU.

4.4 Lock-Step Mode

We created a *lock-step* mode, which builds on QEMU's *icount* mode to synchronize the execution of the VM and simulated hardware. The icount mode disables hardware virtualization support, falling back to an instruction-set simulator with IPC 1 (one instruction per cycle), and modifying the VM's notion of time to be relative to the number of instructions executed (e.g., two billion instructions corresponds to one second of execution). Our variant of this mode further restricts QEMU such that it runs in sync with the HDL simulator, allowing the two to perceive the same notion of time.

The lock-step mode allows co-simulating timing-sensitive interactions between the server software and hardware design. In this mode, the HDL simulator includes an additional unidirectional ZMQ channel to transmit a *clock* message to QEMU on every positive clock edge. We modified the QEMU interpreter loop to perform a blocking read on this channel after QEMU executes the number of instructions corresponding to a single cycle of the hardware design. For example, when targeting a 2GHz server CPU and 250MHz FPGA hardware design, each clock message allows QEMU to advance by eight instructions. To maintain high performance, QEMU internals sometimes cause simulation to advance by more than one instruction before re-entering the interpreter loop. To account for this, our implementation keeps track of the actual number of instructions executed by QEMU and adjusts the number of instructions that are permitted to execute on the subsequent clock message. As a result, any deviation between the server's and the HDL simulator's notion of time is eliminated as soon as it is detected.

The lock-step mode is orders of magnitude slower than the untimed co-simulation running with hardware virtualization. Booting the server in lock-step mode would take multiple days. To make this mode practical for debugging, we support dynamically toggling lock-step execution on a running QEMU instance. Lock-step mode can be disabled to allow QEMU to run without waiting for the HDL simulator while booting the guest operating system and while the developer works on setting up the debugging experiment, and enabled immediately before the start of the experiment.

Table 1: Co-Simulation and Hardware FPGA Platform

Target FPGA Board	NetFPGA SUME (xc7vx690tffg1761-3)
Co-Sim Host	Xeon E5-2620v3, 64GB DDR4
FPGA Compilation Host	Xeon E5-2620v3, 64GB DDR4
Operating System	Ubuntu 16.04, Linux 4.4.0 with KVM
Hypervisor	QEMU 2.7.50
FPGA Tool	Xilinx Vivado 2017.1
HDL Simulator	Synopsys VCS J-2014.12-SP3-8 (with GCC 4.4)
Message Passing Library	ZeroMQ 4.2.1

5 EVALUATION

This section presents an evaluation of our VM-HDL co-simulation framework. For this evaluation, we developed two test cases: an FPGA accelerator for sorting of data, and an FPGA network card device. Using these test cases, we demonstrate how the co-simulation framework allows full-system simulation including hardware, application software, device driver, and operating system, and we evaluate how the co-simulator improves the developer's design and debug experience, in terms of the debug iteration time and the visibility into the internal state of the hardware and software.

5.1 Methodology

We list the details of the hardware and software platform we use for our evaluation of both test designs in Table 1. We intentionally use the same hardware for the co-simulation framework measurements as for the hardware FPGA platform, and use the same versions of the operating system and all software in both.

Sorting Accelerator Design. The sorting accelerator design represents a common style of coarse-grained accelerators. Such hardware designs typically comprise one or several compute units for processing data and a DMA engine to perform data transfers. The software running on the server prepares the input data and triggers the accelerator's DMA and compute unit. The DMA engine fetches input data from the server memory and sends it to the compute unit. After the compute unit finishes processing the input data and generates the result, the DMA engine stores the result back to the server memory and notifies the application software.

We automatically generate the sorting unit using the Spiral Sorting Network IP Generator [24]. The sorting unit takes a stream of input data and produces a stream of output data after a fixed number of cycles. Xilinx DMA IP is used in *basic* mode to fetch input data from the server memory through PCIe, stream data through the sorting unit, and write the results back to the server memory.

Network Card Design. The network card device is an example system that connects the FPGA hardware to a network interface. Compared to the sorting accelerator, the NIC has a more complex dataflow, including finer-granularity interaction with the server operating system. The NIC design uses the Xilinx DMA IP to transfer packets to/from the VM memory, but the controller is configured in *scatter/gather* mode. When sending data to the network, the device driver prepares the packets in server memory and triggers the DMA to fetch them through PCIe and stream them to the FPGA MAC's

Table 2: Run time comparison for operations (μs)

	Hardware Platform	Untimed Co-Sim
MMIO Read	0.74	42,400
DMA and Sorting	23.33	2,830,000

transmit interface. In the hardware FPGA platform, the MAC interface then sends data to the wire; in the co-simulation environment, the NIC FPGA simulation bridge will instead transfer this data to the pseudo device in QEMU. When receiving data, the FPGA MAC's receive interface streams the packets into the DMA unit, where they are transferred to the server memory. After a transfer finishes, the DMA sends an interrupt to the operating system to notify it.

5.2 Full-System Performance

PCIe Bridge Performance. To evaluate the performance of our PCIe simulation bridge, we timed the same operations running on a hardware FPGA platform and in co-simulation. For MMIO, we performed dependent serialized MMIO reads from a block RAM in the hardware design. For DMA, we measured the execution time of the sorting accelerator task, including DMA transfers.

Table 2 shows the performance of our framework compared to the hardware FPGA platform. As expected, the co-simulation runs slower than the hardware FPGA platform, because the co-simulation performs cycle-accurate HDL simulation. By comparing the simulation run time with and without the VM-HDL communication channels, we found that communication with the VM does not noticeably impact performance. The VM runs on a separate host CPU core from the HDL simulation, and polling of the ZMQ channels takes negligible CPU time. Although the poor HDL simulator performance requires developers to still be cautious regarding long test cases while debugging using the co-simulation platform, the performance impact of doing HDL simulation as part of the co-simulation framework is small and is well worth the benefits.

Untimed Mode Time Dilation. When using our co-simulation framework in untimed mode, the QEMU VM runs at bare-metal speed, and time in the VM equals wall-clock time. However, the HDL simulator is running cycle-accurate simulation and is slower than a hardware FPGA platform. Because QEMU and the HDL simulator run independently, the user observes a time dilation between the VM and the hardware design. The performance measurements in Table 2 show this effect to be approximately five orders of magnitude; a hardware design at 250MHz appears as though it is running at approximately 2.5KHz to the software in the VM. However, despite the time dilation, the system remains completely functional and can be used for interactive development and debugging.

Lock-step Mode Performance. To overcome time dilation for timing-sensitive simulations, we utilize the lock-step mode of our framework to force a realistic clock ratio between the VM and the hardware design running inside the HDL simulator.

To measure the accuracy of the co-simulation in lock-step mode, we target a CPU frequency of 2GHz at an IPC of 1 (QEMU *shift=1* setting, which instructs QEMU to treat two billion instructions as equivalent to one second). We target a 250MHz hardware design to set the lock-step multiplier, which means that every HDL simulator clock cycle permits the QEMU VM to advance by eight instructions.

We used a simple *sleep* application to measure time in the virtual machine. We observed that, although the wall clock elapsed time is approximately three orders of magnitude longer than the requested sleep time, the elapsed time observed by both the VM and HDL simulator match the requested sleep time. This indicates that, although the simulation runs much slower than real time, the VM's notion of time remains self-consistent. We also tested our network card design in lock-step mode. We used ping to test the network latency between the VM and its host. Unlike the untimed mode, which resulted in higher latency than on a real system due to the VM observing time faster than HDL simulation, the reported latency in lock-step mode is very similar to a real system. This happens because the time taken by ping outside the co-simulation environment appears instantaneous to the co-simulation (just like when a real host pings itself), resulting in the latency reported by the VM corresponding to the actual number of cycles that elapsed in the lock-step co-simulation framework. This further demonstrates that the notion of time is self-consistent between the VM and the HDL simulator in lock-step mode.

NIC Bridge Performance. Our framework presents new opportunities for debugging FPGA hardware designs with network connectivity by exposing them to the LAN and Internet traffic. However, network packets between real hosts have a higher packet rate than expected by the co-simulation setup, and can overwhelm the co-simulation when processing the test traffic. We therefore use firewall rules on the co-simulation host to filter out background ARP chatter and packets not sent to the co-simulated host. We then use *HTTP file transfer* to evaluate the bandwidth and *ping* to evaluate the latency of network traffic from the co-simulation environment when running network card hardware in HDL simulation.

In the untimed mode, our experiments show that the platform can sustain 15KB/sec connections. The platform introduces an extra 80ms round-trip latency on each ping, indicating that the network card hardware in simulation takes approximately 40ms to process a packet. These results demonstrate that the platform is fast enough to sustain network connectivity to to the real world, without the remote end timing out or retransmitting packets, despite the packets passing through a cycle-accurate HDL simulation of the network card. We also performed these tests in lock-step mode. Due to significant slowdown of the virtual machine, the round-trip ping times observe an additional 400ms of latency. Network transfers from real-world hosts become impractical, as the co-simulation environment falls too far behind to send TCP acknowledgments in a timely fashion and the remote end closes the connection.

5.3 Debug Iteration Time

Hardware Design Changes. While developing and debugging applications, developers frequently need to modify the hardware design due to bugs, design changes, or simply to observe the effects of different design decisions. If working with the hardware FPGA platform, this would require the developer to run the FPGA synthesis and place-and-route process. However, in our co-simulation framework the developer only needs to launch the simulation. We quantify this difference using our sorting accelerator. As shown in Table 3, when compared to a hardware FPGA platform, which takes

Table 3: Run-time comparison (minutes:seconds)

	Hardware Platform	Co-Simulation
Launch Simulator	-	1:10
Synthesis	18:03	-
Place and Route	35:40	-
Reboot	2:33	0:25
Execution	≈0	0:02.8
Total	56:16	1:38

Table 4: System boot time comparison (minutes:seconds)

	Local disk	iSCSI	VM
BIOS	0:55	1:10	0:05
OS	0:33	1:23	0:14
Total	1:26	2:33	0:19

about an hour to go through the FPGA synthesis and place-and-route process, the co-simulation framework can achieve an over 30x reduction in iteration time. Changes in the FPGA platform can run in the co-simulation framework in just a few minutes.

Software Reboot. When debugging accelerators on a hardware FPGA platform, instability in the hardware, software, and device driver can all frequently hang the entire system, requiring slow and tedious reboots. An advantage of our co-simulation framework is that VMs are typically faster to reboot. To illustrate this, Table 4 compares: the boot time from a local disk, the same server booting from an iSCSI disk, and the VM in our co-simulation framework, all running the same version of Linux. The results indicate that the VM reboot time is consistently faster than server reboot.

Additionally, the developer can use VM snapshots to achieve nearly-instant reboot. Although server reboot time seems negligible compared to the FPGA synthesis and place-and-route, frequent and slow system reboots greatly contribute to the slow debug process.

5.4 Full Hardware Visibility

Adding Logic Analyzer Probes. When debugging on a hardware FPGA platform, developers frequently use embedded logic analyzers such as Xilinx ILA [20] or Altera SignalTap II [1] to observe the values of internal signals over time. This approach places practical limitations on the number of signals observed and the number of cycles of data that can be recorded. Naturally, these limitations mean that developers gradually adjust the set of signals they monitor during the debug process. Each time the set of monitored signals changes, the developer must re-run at least place-and-route.

We use our sorting accelerator as a case study to quantify the cost of re-configuring the embedded logic analyzer (Xilinx ILA). For the hardware FPGA platform, we follow a typical debug workflow, where the developer iterates several times, changing the locations of the limited embedded logic analyzer probes each time. We first configured the logic analyzer to observe the ports of the sorting unit (408 pins) for 8192 cycles. We then synthesized and implemented the design, as the developer would need to do to program the FPGA. In the next step, we added one AXI port (659 pins) to the monitored signals and repeated the process. In the third step, we swapped monitoring one AXI port for another one (806 pins). To compare

Table 5: Visibility overhead (minutes:seconds)

	1st Change	2nd Change	3rd Change
Hardware FPGA Platform:			
Synthesis	17:48	-	-
Place and Route	39:16	42:22	58.52
Reboot	1:26	1:26	1:26
Execution	≈0	≈0	≈0
Total	58:30	43:48	60:18
Co-Simulation:			
Launch Simulator	1:10	-	-
Reboot	0:19	-	-
Execution	0:04.2	-	-
Total	1:33	0	0
Speedup	over 30x	∞	∞

with the time required to observe the same signals using our co-simulation framework, we simply configured the HDL simulator to store all signals from the start to the end of the entire simulation.

Table 5 compares the time required for these changes in the two scenarios. On the hardware FPGA platform, the first insertion of the debug core requires both synthesis and place-and-route, which requires nearly an hour even for this relatively small design. Further changes to the monitoring signal set do not require re-synthesis, but the place-and-route time increases along with the number of the signals being observed. In contrast, our co-simulation framework has full visibility into the hardware design after only one simulation run because all signal values in the design are stored. The co-simulator's first iteration is already over 30x faster than the hardware FPGA platform; further iterations are unnecessary.

Cost of Full Visibility. By storing the entire waveform history for every signal in the hardware design, it is possible for our co-simulation framework to enable full visibility of all internal values in the FPGA from a single simulation run. There are two costs associated with storing this large number of signals: the time overhead of writing the data to disk, and the amount of storage required.

To quantify the runtime differences, we compare the co-simulation execution times in Table 3 (which do not record internal signals) and Table 5 (which saves all signals for the entire execution). We observe a 50% increase in execution time when saving signals. As shown in Table 5, this performance overhead is still much lower than the cost of having to re-run place-and-route to reconfigure the hardware FPGA platform's embedded logic analyzer probes. The other cost is the disk space. Experiments shows that a two-hour (wall clock time) simulation of our network card design creates an 8GB waveform file in the FSDB format, suggesting that it is affordable with today's storage capacity even for long simulations.

6 RELATED WORK

HW-SW Co-Sim. Several frameworks introduced by academia, EDA companies, FPGA vendors, and cloud service providers with FPGA offerings aim to co-simulate FPGA hardware and application software. These systems, such as the Message-passing Simulation Framework (MSF) [13], Intel OpenCL for FPGA [5], Intel AFU Simulation Environment (ASE), Xilinx SDAccel [18], and Amazon F1 [3]

allow software testbenches to drive HDL simulation environments. However, these systems are limited to executing application software, rather than allowing full-system co-simulation. In contrast, our framework supports co-simulating and debugging the operating system, device drivers, application software, and hardware designs. Like our approach, the open-source VPCIe co-simulation project [15] uses QEMU to support full-system device driver development in the co-simulation environment. However, while similar in spirit, VPCIe is a proof-of-concept system that significantly restricts the hardware designs that can be simulated and uses interfaces that require extensive modification to QEMU and to the hardware design to allow them to work in co-simulation.

Full-System Simulation for SoC ASICs. Another class of related work targets simulation and design exploration for system-on-chip (SoC) ASICs. In this situation, designers' needs and motivations are quite different than those of developers targeting FPGA-accelerated datacenters. In this context, designers typically aim to study design trade-offs or perform early software development alongside high-level models of hardware systems. For example, [9, 11, 14, 16, 21] use QEMU as an instruction-set simulator, connecting it to high-level models of virtual platforms written in SystemC, and there are commercial tools like [10] that can do full-system simulation with hardware designs in HDL. However, all these platforms generally focus on ARM-based SoC ASICs using early-stage hardware models. This contrasts starkly with the approach and goals of our system, where we require full cycle-accurate simulation of the exact hardware design because we aim to analyze and debug the exact hardware and software that will run in the production environment on the target hardware FPGA platform.

7 CONCLUSIONS

FPGAs hold great promise as accelerators in datacenters; in recent years, we have seen several large-scale deployments of FPGA-accelerated servers. Although the performance and energy advantages of FPGAs are well known, a major challenge to wide-spread use is the difficulty of designing, debugging, and integrating FPGA accelerators. Better methodologies and tools, which can reduce the impact of these obstacles, are crucially needed to improve developer productivity and increase adoption of FPGA accelerators.

In this work we aim to improve a challenging drawback in the typical FPGA accelerator workflow: namely, that testbenches are insufficient for testing and debugging full server systems, but debugging on hardware FPGA platforms is slow and cumbersome, due to the long synthesis and place-and-route process, frequent and tedious system reboots, and insufficient visibility. The combination of these problems results in a time-consuming development process, hindering the effective use of FPGA-equipped servers.

Our VM-HDL co-simulation framework leverages existing and widely used mature technologies: virtual machines (which allow fast execution of the operating system, device driver, and application software), and commercial HDL simulators (which provide cycle-accurate simulation of the FPGA design). We join these together by designing new pin-compatible bridge IP for the FPGA's PCIe and NIC interfaces, allowing developers to move seamlessly between the co-simulation framework and hardware FPGA platform, with identical hardware designs, software, and operating system code. By avoiding the FPGA synthesis and place-and-route process, our framework can drastically reduce the debug iteration time, while providing full visibility of the entire system by enabling the use of standard software debuggers and by comprehensively recording waveforms for all hardware signals. The end result is a co-simulation framework that enables rapid development of FPGA accelerators in datacenter systems.

REFERENCES

[1] Altera. 2017. *Quartus Prime Standard Edition Handbook.*
[2] Amazon 2017. Amazon EC2 F1 FPGA Instances. (2017). https://aws.amazon.com/ec2/instance-types/f1
[3] Amazon 2017. Amazon EC2 F1 FPGA Simulation Environment. (2017). https://github.com/aws/aws-fpga/blob/master/hdk/docs/RTL_Simulating_CL_Designs.md
[4] 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. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–13.
[5] Intel 2015. Intel OpenCL for FPGA. (2015). https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
[6] Intel 2016. Intel-Altera Heterogeneous Architecture Research Platform Program. (2016). https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf
[7] Intel 2017. Intel AFU Simulation Environment. (2017). https://opae.github.io/docs/ase_userguide/ase_userguide.html
[8] KVM 2016. Kernel Virtual Machine (KVM). (2016). https://www.linux-kvm.org/
[9] J. W. Lin, C. C. Wang, C. Y. Chang, C. H. Chen, K. J. Lee, Y. H. Chu, J. C. Yeh, and Y. C. Hsiao. 2009. Full System Simulation and Verification Framework. In *2009 Fifth International Conference on Information Assurance and Security*, Vol. 1. 165–168.
[10] Mentor Seamless 2016. Mentor Seamless. (2016). https://www.mentor.com/products/fv/seamless/
[11] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. 2007. Mixed SW/SystemC SoC Emulation Framework. In *2007 IEEE International Symposium on Industrial Electronics.* 2338–2341.
[12] A Putnam, A M Caulfield, E S Chung, D Chiou, K Constantinides, J Demme, H Esmaeilzadeh, J Fowers, G P Gopal, J Gray, M Haselman, S Hauck, S Heil, A Hormati, J Y Kim, S Lanka, J Larus, E Peterson, S Pope, A Smith, J Thong, P Y Xiao, and D Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA).* 13–24.
[13] M Saldana, E Ramalho, and P Chow. 2008. A Message-Passing Hardware/Software Co-simulation Environment to Aid in Reconfigurable Computing Design Using TMD-MPI. In *2008 International Conference on Reconfigurable Computing and FPGAs.* 265–270.
[14] S. T. Shen, S. Y. Lee, and C. H. Chen. 2010. Full system simulation with QEMU: An approach to multi-view 3D GPU design. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems.* 3877–3880.
[15] VPCIe 2013. vpcie: virtual PCIE devices. (2013). https://github.com/texane/vpcie
[16] Chen-Chieh Wang, Ro-Pun Wong, Jing-Wun Lin, and Chung-Ho Chen. 2009. System-level development and verification framework for high-performance system accelerator. In *VLSI Design, Automation and Test, 2009. VLSI-DAT'09. International Symposium on.* IEEE, 359–362.
[17] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.
[18] Xilinx 2015. Xilinx SDAccel Development Environment. (2015). https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[19] Xilinx 2016. Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit. (2016). https://www.xilinx.com/products/boards-and-kits/vcu118.html
[20] Xilinx 2017. Integrated Logic Analyzer (ILA). (2017). https://www.xilinx.com/products/intellectual-property/ila.html
[21] Tse-Chen Yeh and Ming-Chao Chiang. 2012. On the interfacing between QEMU and SystemC for virtual platform construction: Using DMA as a case. *Journal of Systems Architecture* 58, 3 (2012), 99 – 111. http://www.sciencedirect.com/science/article/pii/S1383762112000045
[22] ZeroMQ 2017. ZeroMQ. (2017). http://zeromq.org
[23] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro* 34, 5 (2014), 32–41.
[24] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. 2016. Streaming Sorting Networks. *ACM Transactions on Design Automation of Electronic Systems* 21, 4 (2016), 55.