2024

Component design for application-directed FPGA system generation frameworks

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

COMPONENT DESIGN FOR APPLICATION-DIRECTED FPGA SYSTEM GENERATION FRAMEWORKS

by

SAHAN LAKSHITHA BANDARA

B.Sc., University of Moratuwa, 2015
M.S., Boston University, 2019

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2024

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Rabia Yazicigil, PhD
Assistant Professor of Electrical and Computer Engineering
Assistant Professor of Biomedical Engineering

Third Reader

Richard West, PhD
Professor of Computer Science

Fourth Reader

Ahmed Sanaullah, PhD
Principal Research Software Engineer
Red Hat Research

Acknowledgments

First, I would like to thank my advisor, Professor Martin C. Herbordt for his valuable insights, guidance, and support throughout my PhD journey. I was always able to reach out to Professor Herbordt for advice, not only after I joined the CAAD Lab, but ever since I first moved here for my master's program back in 2017. I'm especially grateful that he was willing to accept me to the research group one year into my PhD program.

Secondly, I thank my committee members, Prof. Richard West, Prof. Rabia Yazicigil, and Dr. Ahmed Sanaullah for their valuable advice, feedback, and precious time. A special thanks goes to Dr. Ahmed Sanaullah for leading the DISL project on which this dissertation is based. I'd also like to thank my collaborators and co-authors in different research efforts throughout my academic journey; Dr. Chunshu Wu, Ulrich Drepper, Anthony Ducimo, Zaid Tahir, Dr. Anqi Guo, Dr. Pouya Haghi, Prof. Tong Geng, Dr. Vipin Sachdeva, Dr. Woody Sherman, Noah Cherry, Dr. Chen Yang, Dr. Rashmi Agrawal, Dr. Alan Ehret, Dr. Mihailo Isakov, Miguel Mark, and Donato Kava for their contributions, expertise, and insights, which have significantly enhanced the quality of my research. I would like to express my special appreciation to my research advisor during the master's program and the first year of my PhD, Professor Michel A. Kinsy for encouraging me to pursue a PhD and continuing to be available for me to reach out for advice any time.

Next, I would like to thank my wife, Hasini for all the support she has provided since I started my academic journey all while completing her PhD and currently working as a postdoctoral researcher. She has always encouraged me to aim higher and do my best. I also like to thank my parents, brother, and other family members for supporting me in various ways during the last seven years.

Finally, I like to express my appreciation for my friends at Boston University and elsewhere; Ranga, Binu, Nari, Rashmi, Alan, Donato, Mihailo, Miguel, Chunshu, Pouya, Anqi, Hafsah, Reza, Zaid, Xiteng, Shining, Chathura, Tony, Rushi, Robert, Zahra, Penny, Efe, Cansu, Nadee, Nipuna, Maneesha, Tharanga, Maleen, Sandamalee, Sachille, Namitha, Prashan, Dinelka, and numerous others I did not mention by name, for their continued friendship even though I am not the most responsive or available when it comes to being friends.

COMPONENT DESIGN FOR APPLICATION-DIRECTED FPGA SYSTEM GENERATION FRAMEWORKS

SAHAN LAKSHITHA BANDARA

Boston University, College of Engineering, 2024

Major Professor: Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

Field Programmable Gate Arrays (FPGAs) can fulfill many critical and contrasting roles in modern computing due to their combination of powerful computing and communication, inherent hardware flexibility, and energy efficiency. FPGAs are traditionally used in application areas such as emulation, prototyping, telecommunication, network packet processing, Digital Signal Processing (DSP), and a myriad of embedded and edge applications. Over the last decade, this use has expanded to include various functions in data centers including supporting low-latency communication and as a computing resource offered by cloud service providers.

There are, however, challenges in development and design portability in FPGAs as the typical design flows involve rebuilding the entire hardware stack for each deployment. To overcome these challenges and make FPGAs more accessible to developers, FPGA vendors and academic researchers have made attempts to add operating system-like abstractions to the FPGA use model. One approach is providing infrastructure logic, typically referred to as an FPGA shell, that implements and manages external interfaces and provides services necessary for application logic to function properly. While they simplify the FPGA use model, fixed implementations of FPGA

shells do not fully address the design portability limitations. They often use FPGA resources unnecessarily as most applications do not require all the capabilities of the FPGA shell, and there is no flexibility in terms of the features implemented by the FPGA shell.

Automatic generation of FPGA system designs based on application requirements can overcome the limitations of fixed FPGA shells. It allows the infrastructure logic to be customized to match the application requirements and, therefore, to provide better resource utilization. Automatic system generation also makes it easier to port designs across devices. We refer to a system design that manages FPGA resources and provides services to a user application as a "hardware operating system" (hOS); and a framework that maps user requirements and available system components to such system designs as an **hOS generator**.

Critical to automatic system generation for FPGAs are system components designed to be integrated into an hOS generator. In this dissertation, we develop a design strategy that maximizes component reuse and design portability while maintaining the implementation effort at an acceptable level. We also present a component design example that follows the proposed design strategy to implement a host-FPGA PCIe communication subsystem. We demonstrate how the PCIe subsystem is integrated into a system generator framework, used to enable different applications, and ported to different devices.

Additionally, we establish a set of characteristics for a good hOS generator design. We also discuss how and to what extent the system generation framework used in this work, named DISL, displays these ideal characteristics. Finally, we attempt to address the open question of how to evaluate a system generator. We discuss qualitative metrics and how they relate to the previously identified ideal characteristics of an hOS generator; and evaluate DISL based on these metrics.

Contents

List of Tables

List of Figures

List of Abbreviations

ARP	.....	Address Resolution Protocol
AXI	.....	Advanced eXtensible Interface
BAR	.....	Base Address Register
BIOS	.....	Basic Input/Output System
CAD	.....	Computer-Aided Design
COTS	.....	Commercial Off-The-Shelf
CPU	.....	Central Processing Unit
DPU	.....	Data Processing Unit
DSP	.....	Digital Signal Processing
FPGA	.....	Field Programmable Gate Array
FSM	.....	Finite State Machine
GUI	.....	Graphical User Interface
HDL	.....	Hardware Description Language
HLS	.....	High Level Synthesis
IPU	.....	Infrastructure Processing Unit
LUT	.....	Look Up Table
NIC	.....	Network Interface Card
NPU	.....	Neural Processing Unit
PCIe	.....	Peripheral Component Interconnect Express
SERDES	.....	Serializer/Deserializer
TCL	.....	Tool Command Language
TLP	.....	Transaction Layer Packet
UART	.....	Universal Asynchronous Receiver Transmitter

Chapter 1

Introduction

1.1 Context

From the 1970s to the 2000s, CPU makers depended mainly on improvements in single-core performance for increasing computing power. These improvements were due to a number of factors: architectural advances, often as a function of newly available chip area; architectural advances, such as deeper pipelines, that allowed for increased operating frequency; and, most of all, process-driven increases in operating frequency due to reductions in feature size. So important was the latter that, between major nodes, there were often complete cycles that heavily relied on shrinking of the process technology for performance improvement with minimal microarchitecture changes [Intel, 2011, Wikipedia, 2024c]. In the early 2000s, when frequency scaling became more challenging, architectural advances continued with increased parallelism on instruction, data, and task levels, with deeper pipelines, superscalar processors, and multi-core architectures. Today Moore's Law [Moore, 1965, Wikipedia, 2024b] and Dennard scaling [Dennard et al., 1974, Wikipedia, 2024a] are close to the end of being able to extract more computing power only from single-core performance improvements. High energy density and resulting dark silicon are putting a damper on multi-core performance as well. CPU makers are forced to focus on energy efficiency with architectures such as ARM big.LITTLE [ARM, 2024] and Intel Performance Hybrid Architecture [Intel, 2023b]. The demand for more computing power from modern workloads such as machine learning is becoming increasingly more difficult

for traditional processors to handle and is currently addressed, e.g., with general-purpose accelerators such as GPGPUs. However, there are challenges in scaling across multiple devices efficiently as some of these workloads cannot be supported by a single device type.

More recently there is a movement towards specialization with domain-specific architectures rather than general-purpose processors and accelerators. Some current examples of domain-specific architectures are for machine learning [Google Cloud, 2024b, Abts et al., 2022, Prabhakar et al., 2022]. Even general-purpose accelerators, such as GPUs and FPGAs, contain ever more specialized hardware to support specific application domains [Nvidia, 2024, AMD, 2024a]. During their Turing Lecture [Hennessy and Patterson, 2019], David Patterson and John Hennessy called this "A New Golden Age of Architecture," and, among other trends, described the movement back to specialization and domain-specific architectures.

Specialization, however, has its limits. We cannot build custom chips and accelerators, or allocate area on general-purpose chips (as we do for common operations such as encryption), for every possible type of specialized operation/application. This is where reconfigurability comes into play. There are domain-specific architectures with a certain level of reconfigurability, enabling those to be configured to match different applications from the same domain, e.g., data-flow architectures [Prabhakar et al., 2022]. There is also interest in integrating these components with coarse-grained reconfigurability to general-purpose processors [AMD, 2024c, AMD, 2024b].

While there is increased interest in coarse-grained reconfigurability, including through Coarse Grained Reconfigurable Arrays (CGRAs), Field Programmable Gate Arrays (FPGA) have been around since the 1980s providing fine-grained reconfigurability. Originally developed to provide glue logic in a single package, their complexity grew along Moore's Law until their applicability had broadened to include a number of

stand-alone use cases. These have included emulation, prototyping, communication, Digital Signal Processing (DSP), embedded applications, and, above all, applications where tight coupling of communication and computation was paramount. With the push for more reconfigurability and customization, they are becoming more commonplace from the data center to the edge, and are used in many different configurations. With their fine-grained reconfigurability, FPGAs can act as accelerators for multiple application domains.

Another recent trend, especially in the data center, is offloading certain tasks to external devices to free up more CPU cycles for user workloads. These are referred to (by different vendors) as DPUs [Burstein, 2021], IPUs [Sundar et al., 2023], or smartNICs [Dastidar et al., 2023]. These devices provide a level of programmability that allows their use to help applications scale well over multiple nodes. FPGAs are a commercial off-the-shelf (COTS) solution that match all of these scenarios and are becoming popular in data centers.

FPGAs can fulfill many contrasting roles in modern computing due to their combination of powerful computing and communication, especially when these are tightly coupled; inherent hardware flexibility; and energy efficiency. However, while the number of FPGAs deployed from the cloud to the edge is on the rise, FPGAs remain challenging to program compared to CPUs and even other accelerators such as GPUs. Despite immense advances in FPGA programming environments, including the High-Level Synthesis (HLS) tools, the level of hardware knowledge required to program an FPGA often still makes it prohibitive for most application developers to utilize the FPGAs to accelerate their applications. FPGAs are becoming increasingly heterogeneous and coarse-grained with more hardened logic added alongside and within the reconfigurable fabric. These can range from fine-grained components within the FPGA fabric, such as block RAMs (BRAM) and Digital Signal Processing (DSP)

slices to larger ones such as Gigabit transceivers, and PHYs for external interfaces such as PCIe and DRAM. These hardened logic blocks enable higher operating frequencies, higher I/O bandwidth, and better resource utilization. However, effectively incorporating these into designs takes additional effort and specialized knowledge compared to implementing simple application logic.

HLS tools have taken significant leaps in recent years. These allow application developers to use high-level languages such as C or Python to describe hardware instead of hardware description languages (HDL). High-level languages together with HLS tools are becoming increasingly good at implementing application logic, sometimes providing comparable performance to HDL designs.

1.2 The Problem Addressed

The problem addressed in this dissertation begins with the fact that a functioning FPGA design usually requires infrastructure logic such as I/O to facilitate the functionality of the application logic. Implementing I/O interfaces typically involves logic blocks that are *hard* in the sense that they are not configurable. These hard blocks vary significantly across devices, in functionality and the types of interfaces exposed. Incorporating these blocks into designs requires specialized knowledge. Describing the infrastructure logic in a high-level language and generating it using HLS is neither efficient nor straightforward. Therefore, rather than requiring a user to design this infrastructure logic, FPGA vendors provide FPGA *shells*, which take up a fixed region on the FPGA fabric to implement all the infrastructure logic to enable the application logic designed by the user.

These vendor-provided shells, however, have several limitations.

- The fixed implementations take up a fixed amount of FPGA resources *even if some of the features/interfaces are not used by a given design.*

- FPGA shells are not available for all devices, limiting users to a fixed set of devices/development boards.
- FPGA shells require the user logic to implement specific interfaces to interact with the shell. This inhibits flexibility to optimize and to alternative design strategies: users are forced to follow the model/interface/connectivity the vendor believes to be optimal.
- FPGA shells, including their interfaces, vary by device, even within a device family. This often requires applications be updated, sometimes through substantial engineering effort, whenever an application is ported to a new device.

Expanding on this last point, even when an expert implements an FPGA design, the design portability is low due to the device- and vendor-specific aspects of the design. Application logic typically described in a hardware description language or a high-level language targeting high-level synthesis is generally at least somewhat portable across devices. However, additional design complexity and lack of portability arise from using device- and vendor-specific components.

Moving beyond the FPGA chips themselves, commercially available FPGA development boards each have different peripheral components – for instance, the type of DRAM or the type of transceivers for network connectivity. A designer is required to capture all the device/vendor-specific details in their design. Typical FPGA design flow involves redesigning most of the hardware stack whenever designing a new application or porting an existing application to a different device. *This is highly inefficient and the level of effort and expertise required create high barriers to entry and design reuse.*

We posit in this thesis that the current state of FPGA development is analogous to how the CPUs were programmed before portable operating systems: an application developer might need to learn and integrate an entirely new API to port an application

to a new architecture, even from the same vendor. As FPGAs do not have operating systems, each application design also entails implementing, or at least accounting for, the entire hardware stack.

Prior work has attempted to enhance the usability of FPGAs by introducing operating system-like abstractions to the FPGA use model. There are existing offerings that behave like an operating system for FPGAs by implementing external interfaces and providing services to user logic over fixed interfaces. We refer to such an implementation as a **hardware operating system** (hOS) due to their similarity to a software operating system in managing system resources and providing services to applications. These include FPGA shells from FPGA vendors and third parties. This terminology is elaborated further in Section 1.4.

Device support for current hOS offerings is extremely limited regardless of whether they are provided by device vendors or third parties. The vendor- and device-specific nature of FPGA shells, and the lack of clear separation between device-specific and generic logic, makes it impossible to port an FPGA shell designed for a particular device to another without incurring significant engineering effort to reimplement most of the components. Since each FPGA shell offers a different set of interfaces and services, porting a design from one FPGA shell to another is also challenging and would possibly require parts of the design to be reworked. Another limitation of existing offerings is the lack of flexibility in terms of features. A user has limited control over which components are included in the FPGA shell regardless of the set of requirements for a given user application.

Some of the issues described above can be mitigated if there is an intermediate layer or framework that maps the various application requirements to different system configurations. Figure 1·1 depicts this idea. Each of the applications has its own set of requirements. The mapping framework maps each application to a different system

configuration that satisfies the application requirements.

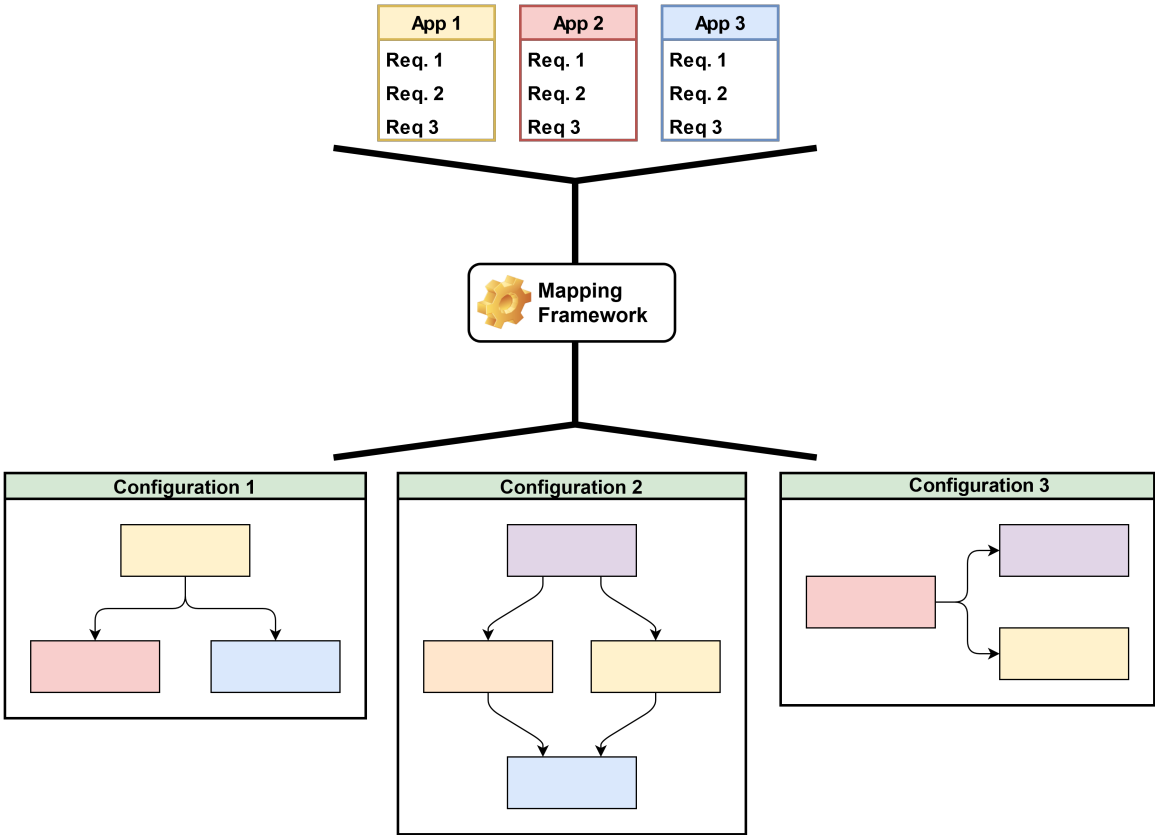


Figure 1·1: Mapping application requirements to system configurations.

For example, assume that *App 1* is a SmartNIC application and *App 2* is an accelerator for some computationally intensive part of a host program. The system configuration for App 1 must include PCIe and network interfaces. However, the configuration for App 2 only needs the PCIe interface. Both configurations could include other interfaces such as DRAM depending on additional application requirements. If the same system configuration is used for both applications, FPGA resources are wasted in the case of App 2 because the network interface is unnecessary. It is the responsibility of the mapping framework to map each application to a system configuration tailored to that application.

The mapping framework in Figure 1·1 maps each of the applications to existing system implementations. This way, we are ultimately going to run into a similar issue as existing solutions where the mapping framework cannot find the ideal configurations for certain applications because those were never implemented. Therefore, it is forced to map these applications to sub-optimal configurations.

Rather than mapping applications to fixed implementations, making the mapping framework also automatically generate the system allows us to map more applications to tailored systems. In this case, instead of existing implementations, the system components will be an input to the intermediate layer that generates the customized system. Such a framework that maps application requirements to a system configuration and generates the said system using components from a component library is referred to as a **hardware OS generator** in this work.

OS-like implementations for FPGAs have severe limitations with regard to supported devices. All such offerings typically have a short list of supported devices. In contrast to this, software operating systems support a larger variety of architectures and devices without the necessity to rework the operating system or the user applications. An hOS generator capable of generating hardware OSs for different devices enables a user to keep the application logic constant and generate different implementations of infrastructure logic to match different devices and deployment scenarios. This reduces the effort necessary to port applications between devices. By implementing a design strategy that clearly separates device-specific and -agnostic logic for hOS components, parts of infrastructure logic can also be reused and ported to different devices.

To enable an hOS generator to achieve the capabilities mentioned above, the components themselves must be designed with those capabilities in mind. Applying the abstract concepts discussed in relation to a system generator to the components is

not a straightforward exercise. For instance, consider the idea of clearly separating the device-specific and generic logic when implementing an hOS. When applying this concept to a component, there is a range of strategies that achieve this goal to different extents. This is especially true for components that involve external interfaces. Given the increasingly heterogeneous nature of the current FPGA architectures, a component used by the hOS generator could include hardened ASIC blocks, IP cores that include both device-specific and -agnostic logic, and generic logic implemented by an FPGA designer. There is a multitude of ways all of these could be separated into device-specific and generic partitions. However, the separation should be carried out keeping the target outcome in mind. An unnecessarily strict partitioning scheme may result in high implementation overhead while not providing any additional benefit in terms of the target capabilities.

1.3 Thesis Statement

Before stating our thesis we summarize the discussion so far.

- Reconfigurable devices such as FPGAs have an essential place in a variety of application domains.
- FPGAs require logic to interface between the internal application and external devices, i.e., an hOS.
- This hOS currently is either general, e.g., supplied by the vendor, or built to order. Both have problems: the first inhibits portability and efficiency, while the second is too expensive for all but the largest applications.
- An hOS *generator* that creates a bespoke hOS for any given device/application combination would solve this problem, but has not yet been successfully created.

A crucial factor in creating an hOS generator is the availability of components that possess certain attributes so that the hOS generated using them achieves the goals of a system generator such as: (i) reducing the expertise and effort required to build FPGA applications, (ii) providing the exact services required by the application logic without overusing resources, and (iii) improving design portability and reusability.

In particular, our thesis is that **if the components used by the system generator have been created following a design strategy that applies the abstract goals of the system generator to the components, then an application-directed system generation framework for FPGAs can reduce the specialized knowledge and effort necessary to implement FPGA designs while also improving resource utilization and design portability.**

1.4 Terminology and Scope

The terms *hardware operating system* and *FPGA operating system* have been used in prior work to refer to various attempts at introducing operating system-like abstractions to the FPGA use model. However, there is no consensus on what an operating system for FPGAs is. The strategies employed in prior work range from modifications to host operating systems to FPGA shells providing services to user applications implemented on FPGAs. The efforts focused on host operating systems/hypervisors typically target FPGA-based accelerators and attempt to integrate them into the thread/process abstractions of the host OS to simplify offloading compute tasks to the accelerators. The modifications to the operating systems/hypervisors could be accompanied by FPGA shells that implement external interfaces and provide the connectivity between the host OS/hypervisor and application logic on the FPGA.

The set of works focused on FPGA shell implementations attempts to provide advanced capabilities via the shell implementation rather than making modifications

to the host operating system. These are accompanied by software components such as runtime libraries, and device drivers on the host side. FPGA shell implementations from FPGA vendors also fall under this category. The capabilities provided by different FPGA shells from prior academic works and vendors range from basic capabilities such as implementing external interfaces to more advanced capabilities such as scheduling different application kernels using partial reconfiguration, to fully software operating system-like capabilities such as virtual memory enabling hardware kernels on the FPGA to share an address space with an application executing on a host machine. There is no clear boundary between these two approaches. Rather, the research focus of each work falls on a spectrum between software and hardware. Most solutions include both hardware and software components.

In this dissertation, we define a hardware operating system as a layer of hardware that implements and manages lower-level interfaces and provides services (such as memory and network access) to the application logic on the FPGA. One or more user applications can share the FPGA resources spatially or temporally. Beyond these basic capabilities, any other advanced functionality such as scheduling FPGA kernels using partial reconfiguration could be implemented as optional capabilities to be selected by a user based on the specific requirements for the given application(s) and deployment context. This approach enables streamlined implementations without the unnecessary overhead incurred by unused features. Most FPGA shell implementations from vendors and third parties match the above definition except for the fact that they are fixed implementations without the flexibility to select the capabilities. Therefore, throughout this dissertation, we use the term **hOS** to refer to portions of generated systems or fixed implementations from prior works that match our definition of an hOS. These include the part of the design referred to using terms such as FPGA shell, and platform layer in prior works.

We do not consider the host software stack a part of the FPGA hardware OS (hOS). However, the hOS should implement the interfaces and any additional logic necessary to interact with the host software chosen by a user. One reason to separate the host software from the hOS is to not limit an hOS to the accelerator use model as many of the prior works have. An hOS should support any of the different deployment scenarios for FPGAs and not all of these involve a host machine controlling the FPGA. Therefore we focus our efforts on implementing the necessary infrastructure layer on the FPGA without specifying the connectivity with a host. In Chapter 4, we present a PCIe subsystem implementation that provides host-FPGA communication. However, it is implemented as a discrete component that can be selected and configured by a user instead of a tightly coupled part of an FPGA hOS.

Much of the prior works on introducing OS-like abstractions for FPGAs focus extensively on application developers who are not hardware developers to the point of trying to hide the hardware details completely from a user by creating deep abstraction layers (eg: virtual memory, memory management) that allow an FPGA to behave very much like a general purpose processor and integrate FPGAs into the host system's software stack. While this results in a simple use model for a software developer, it also lacks the flexibility for a hardware developer to choose capabilities and optimize the infrastructure logic to match a particular application/deployment context. In this work, we consider a hardware OS for FPGAs to be useful for a wider range of users with different levels of expertise in hardware design. Generating hardware OSs specific to a given use case is more suitable to realize the above goal compared to providing fixed implementations that lack flexibility.

We use an existing FPGA system generator named **Dynamic Infrastructure Services Layer (DISL)** as the basis for the component and system implementations presented in this work. Therefore, this dissertation shares some of the terminology

from DISL. DISL is covered in detail as part of related work in Chapter 2.

1.5 Contributions

There are two major high-level contributions in this dissertation.

The first is the exploration of how the abstract design concepts for a hardware operating system generator are applied to the component designs and the development of a design strategy for hOS components that achieves the overall design targets of an hOS generator framework, such as reuse and portability.

The second is a PCIe subsystem that provides host-FPGA communication services to user applications and enables the repurposing of in-kernel Virtio device drivers as generic device drivers for FPGAs. We design and implement the PCIe subsystem following the design strategy identified above. We also integrate the PCIe subsystem into the DISL FPGA system generator framework and demonstrate how the PCIe subsystem enables different applications and how the component design guidelines enable component reuse and design portability.

Some additional contributions are as follows.

We conduct a discussion on hardware operating systems and hOS generators to establish a set of ideal characteristics for each. Next, we explore the relationship between these attributes and the design decisions regarding the hOS design and the components used to generate the hardware operating systems. We also analyze the implementation of DISL and related works to identify how they achieve some of the desirable characteristics.

Also, we address the open question of how to evaluate an hOS generator for FPGAs. Prior work has established different metrics such as resource and performance overhead to evaluate the OS-like abstractions for FPGAs. However, evaluating a system generator is not addressed in prior work. We discuss several qualitative met-

rics and how they relate to the previously discussed ideal characteristics of an hOS generator. We also evaluate DISL using the proposed metrics and propose potential improvements.

These contributions have the potential for the following significance.

1. We develop a set of design guidelines the FPGA community could use when implementing components for system generators. These guidelines are meant to improve the reusability and portability of the components as well as the systems generated using the components.

2. We address a limitation in existing solutions for host-FPGA communication with a PCIe subsystem design that allows the FPGA community to replace vendor-provided and custom device drivers with generic Virtio drivers.

3. By establishing a set of ideal attributes for system generators and exploring their relationship with some of the design decisions, this work assists future researchers in evaluating their design choices for system generators.

1.6 Organization

The remainder of this dissertation is organized as follows.

- We discuss prior work on introducing operating system-like abstractions to FPGAs in Chapter 2.

- In Chapter 3, we establish a set of ideal characteristics for a general-purpose hardware OS generator for FPGAs and assess how well the existing FPGA system generators incorporate these characteristics. Next, we explore the relationship between the hOS generator attributes and the design decisions regarding the hOS components. Finally, we present a set of design guidelines for hOS components.

- A PCIe subsystem design that allows using in-kernel Virtio device drivers as generic device drivers for FPGAs is presented in Chapter 4.

- We evaluate the performance of Virtio drivers against vendor-provided device drivers in Chapter 5. We also port the PCIe subsystem to a new device.

- In Chapter 6, we discuss qualitative metrics to evaluate an hOS generator and evaluate the system generator used in this work using a design example. We demonstrate how a hardware OS generator can be used to implement different applications and easily port them across devices while scaling the application to better utilize the different resource availability of the different devices. Furthermore, we demonstrate how application-specific generation of the hardware OS results in lower overhead compared to a fixed implementation.

- Finally, in Chapter 7, we present future research directions and conclude this dissertation.

Chapter 2

Background and Related Work

In this chapter, we provide background and present the prior work related to the topics discussed in this dissertation. The main sections of this chapter cover the following.

1. Prior work on introducing operating system like abstractions to FPGAs, system generators for FPGAs, and FPGA shells.

2. Description of DISL, which is the system generator used for all the implementations in this work.

3. Prior work on improving host-FPGA communication, related to the component design example in Chapter 4.

4. Prior work related to different applications discussed in this dissertation, including Key-value stores and distributed caching in the network implemented in the example design in Chapter 6.

2.1 OS-like Abstractions, FPGA shells, and System Generators

In this section, we cover prior work on FPGA virtualization and OS-like abstractions for FPGAs. FPGAs do not have operating systems. Therefore, a typical FPGA design cycle involves implementing the complete hardware stack. While this allows

application-specific optimizations, this is not necessary in a lot of use cases. Therefore, many prior works have explored the idea of introducing operating system like abstractions to the FPGA use model. These works attempt to achieve a combination of objectives such as:

- Providing an abstraction that hides the hardware specifics from application developers and provides applications with simple, consistent, and familiar interfaces to access the FPGA resources
- Enabling efficient sharing of FPGA resources among multiple users and applications
- Transparent provisioning and management of FPGA resources
- Data security and system resilience by means of performance and data isolation among different users and applications
- Flexibility and scalability when deploying different applications on FPGAs

These works can be broadly categorized into two groups. The first set of works attempts to integrate FPGA-based accelerators into the thread/process model of the host operating system. These usually involve modifications to the host operating system or the hypervisor. The second set of prior works focuses on providing advanced capabilities such as scheduling within the FPGA itself.

2.1.1 Integrating FPGAs into the host OS/Hypervisor

Berkeley Operating system for ReProgrammable Hardware (BORPH) [So and Brodersen, 2006, So, 2007, So and Brodersen, 2008] is an operating system designed for FPGA-based reconfigurable computers. The authors extend the Linux kernel to provide kernel support for FPGA applications. BORPH establishes the notion of "hardware process" to represent an FPGA application and treat them as regular Linux

processes. Although BORPH targets FPGAs deployed as tightly or loosely coupled coprocessors, it doesn't treat the FPGA as a coprocessor that is part of the hardware platform. Instead, BORPH represents the reconfigurable fabric as a general-purpose computing resource that a user can schedule processes on (similar to a logical core of the CPU). BORPH uses regions of FPGA fabric as a computation unit to spawn hardware processes, similar to the way software processes are spawned to a processor. Each reconfigurable region is defined as a hardware region(hwr). A hardware process is spawned using a BORPH object file which is a binary file format that includes the bitstreams and methods necessary to configure an FPGA region. Communication with the FPGA region is implemented using standard UNIX file pipes.

The authors of [Peck et al., 2006, Anderson et al., 2006], and [Anderson et al., 2007] present the 'Hybridthreads Computational Model' (hthreads) and the 'Hardware Thread Interface' (HWTI) as a unifying programming model for application threads running within a hybrid CPU/FPGA system. The HWTI supports the generalized pthreads API semantics. Threads within an hthreads design are either compiled to run as software binaries on general-purpose CPUs or are translated and synthesized as application-specific hardware threads to run as hardware cores within the FPGA fabric. The HWTI provides a hardware thread with a global distributed memory, support for pointers, a generalized function call model including recursion, local variable declaration, dynamic memory allocation, and a remote procedural call model that enables hardware thread access to library functions.

ReconOS [Lübbers and Platzner, 2009, Agne et al., 2013] is another attempt at semantically integrating hardware accelerators into the host OS environment. It extends the multithreading programming model with hardware thread support. A hardware thread represents an accelerator implemented on the FPGA. An 'OS synchronization FSM' is also implemented alongside each accelerator to enable OS calls

from within the hardware thread. These hardware and software modifications let the hardware threads interact with software threads using standard OS mechanisms such as semaphores, mutexes, condition variables, and message queues.

OPTIMUS [Ma et al., 2020] is a hypervisor that supports a shared memory model for FPGA-based accelerators instead of the typical host-centric model. In a shared memory model, the accelerator design implemented on the FPGA shares a memory space with a process running on the host CPU. It can also issue DMA requests without the involvement of the host CPU. OPTIMUS targets use cases where a service provider programs the FPGA with multiple accelerators and the hypervisor uses spatial and temporal multiplexing to map the user processes to the accelerators. OPTIMUS does not handle the reconfiguration of the FPGA. Authors of [Liu et al., 2021] present 'MEGATRON' an extension to [Ma et al., 2020] aimed at improving the DMA performance of a shared memory FPGA platform. MEGATRON is a hybrid address translation service consisting of a hardware TLB and a software page table walker.

2.1.2 Implementing OS-like capabilities on the FPGA

There are hOS-like solutions provided by FPGA vendors, typically referred to as FPGA shells. An FPGA shell takes up a fixed portion of the FPGA fabric to implement the communication infrastructure such as PCIe and Ethernet controllers, DMA engines, memory interfaces, and any other peripherals. These are typically coupled with runtime libraries such as Xilinx runtime library (XRT) [Xilinx Inc., 2022] and Open Programmable Acceleration Engine (OPAE) [Intel, 2017], which provide the user with simple APIs for programming, data movement, and controlling the FPGA. These typically provide a user-space library, and kernel space device drivers that work with the FPGA shell. However, these FPGA shells are designed to target specific devices and do not differentiate between device-specific and generic logic. Therefore, porting a shell from one design to another is not a trivial task as it involves redesign-

ing the whole shell. Since the device support is very limited, the same design may not be reusable on a different device from the same vendor even when it is a device of the same family with comparable hardware resources.

The cloud service providers such as Amazon [Amazon Web Services, 2024], and Microsoft [Putnam et al., 2016] who offer FPGAs as a service also use FPGA shells in their deployments. They provide custom toolchains that allow a user to compile a user design and ensure that the design is compatible with the FPGA shell. Similar to vendor-provided FPGA shells, the user design is loaded into the dynamic region of the FPGA using partial reconfiguration (PR). The FPGA can be shared among multiple users by loading different bit streams. The toolchains and the FPGA shells are specific to the FPGA boards used by the different cloud service providers. There is no way to deploy the same design on FPGAs from different service providers without significant engineering effort.

The Latency-insensitive Environment for Application Programming (LEAP) [Fleming and Adler, 2016] is built around latency-insensitive communications channels. The authors argue that an FPGA operating system should use a different abstraction from an OS for general-purpose CPUs to represent the fundamentally different computing model of an FPGA. LEAP provides device abstractions for FPGAs and a collection of I/O and memory management services for the applications.

The authors of [Oliver et al., 2011] describes coupling CPUs and FPGAs using Intel QuickPath Interconnect (QPI), which is a cache-coherent fabric used in Intel CPUs. In the platform described in this work, one of the CPU sockets is populated with an FPGA module capable of implementing a QPI link. The application-specific hardware elements instantiated on the FPGA are called 'Accelerator Functional Units' (AFU). The static region implements the QPI physical, link, and protocol layers. The use of a cache-coherent fabric enables many new capabilities such as shared virtual

memory and low-latency message transfers.

The authors of [Vogel et al., 2018] also present a hardware/software framework for enabling shared virtual memory for FPGA accelerators. They target platform FPGAs that are heterogeneous SoCs that includes a hard processor system alongside the FPGA fabric. The main hardware component introduced is a configurable IOMMU design that can be instantiated on the FPGA and interfaced with the accelerator to enable shared virtual memory. The software components running on the host include a kernel-level driver to manage the IOMMU hardware and a user space runtime library.

The Feniks FPGA operating system [Zhang et al., 2017] targets large-scale FPGA deployments in datacenters. It provides abstracted interfaces to FPGA accelerators including host, off-chip memory, and network interface. Feniks uses partial reconfiguration (PR) to separate the operating system and the application accelerators. It also provides a resource allocation framework for FPGAs throughout a datacenter with the assistance of a resource allocation agent running on host CPUs. Application developers are provided with a set of templates with different PR region configurations.

The authors of [Vaishnav et al., 2020] attempt to develop a modular FPGA operating system with *FOS*. They envision each component in FOS both hardware and software being modular and replaceable. The aim is to allow hardware, software, accelerator, and application developers to innovate independently and still being able to combine their implementation together to build functioning designs.

The most feature-rich hOS implementation we are aware of is Coyote [Korolija et al., 2020]. It is a suite of OS abstractions working with the host OS to provide capabilities such as shared virtual memory, communication, and memory management to the hardware accelerators. Nested dynamic reconfiguration is used to support independently reconfigurable hOS layers. The application layer is divided into multiple

'virtual FPGAs' (vFPGA). Accelerators share these PR regions spatially and temporally. The service layer implements external interfaces such as memory and network, and additional services such as TCP/IP, and RDMA. Finally, the static layer manages the other two layers and implements host-FPGA connectivity.

2.1.3 System Generators

The Task Parallel System Composer (TaPaSCo) [Heinz et al., 2021] is an open-source toolflow to generate complete FPGA systems by integrating FPGA-based accelerators into a heterogeneous system. It also provides a simple programming interface. The application developers are required to implement the individual accelerators (referred to as PEs) following a strict interface specification. The PEs are grouped together into a strict clustered architecture. The external interfaces are implemented using a fixed layer referred to as the *Platform*. The *ThreadPoolComposer* presented in [Korinth et al., 2015] is a predecessor of TaPaSCo.

The $ARTICo^3$ framework presented in [Rodríguez et al., 2018] can generate FPGA-based embedded systems targeting edge computing in Cyber-Physical Systems. The framework includes a hardware-based processing architecture, an automated toolchain to generate systems from high-level descriptions, and a runtime to deploy and manage the generated systems on the FPGA. Dynamic partial reconfiguration is used to schedule the accelerators onto the FPGA fabric. RapidSoC [Wenzel and Hochberger, 2016] is a framework to generate softcore processor based SoCs targeting FPGAs.

2.1.4 Design Automation

Essential to any FPGA design project is the use of design automation tools such as High Level Synthesis (HLS) [Hauck and DeHon, 2008, Gokhale et al., 2004]. There is therefore an analogy between HLS and DISL (for hardware) with respect to com-

pilers and operating systems (for software), respectively. Some recent work applying Machine Learning to HLS includes [Shahzad et al., 2022, Munafo et al., 2023, Shahzad et al., 2024b, Sajjadinasab et al., 2024a, Shahzad et al., 2024a].

2.2 Dynamic Infrastructure Services Layer

In this section, we describe the Dynamic Infrastructure Services Layer (DISL). A detailed description is provided because it is the overarching project that includes the work presented in this dissertation. We use DISL for all the implementations in this work and the discussions in later chapters use the terminology of DISL.

DISL is a framework to automatically generate hardware operating systems for FPGAs using component libraries and user requirements as inputs. More accurately, DISL can be described as a system generator because it generates full systems composed of application logic, device-independent infrastructure logic referred to as the hardware OS, and device-specific logic referred to as the BIOS layer. The infrastructure logic is customized to match application requirements and instantiated alongside the application logic in the source file for the top module of the design hierarchy. This is somewhat similar to the unikernel approach in software, where an application is statically linked with a specialized kernel that implements only the functionality necessary for that application. This approach allows DISL to generate application-specific systems and provide portability without generating the different layers independently. Apart from the system to be implemented, DISL also generates the compilation and runtime environments including scripts, Makefiles, and binaries among other supplementary files.

Figure 2·1 presents a simplified overview of the DISL framework. The inputs can be broadly categorized as user requirements and component libraries. User inputs could include descriptions of target applications, including source files for the user

applications, information regarding the deployment context (the components to be included and the connectivity among them), and any system constraints such as Baud rates for serial interfaces, and the target operating frequency. The component libraries could include descriptions of device PHYs, vendor-provided IP cores, and third-party IP cores. DISL combines the user requirements and the components to generate FPGA designs.

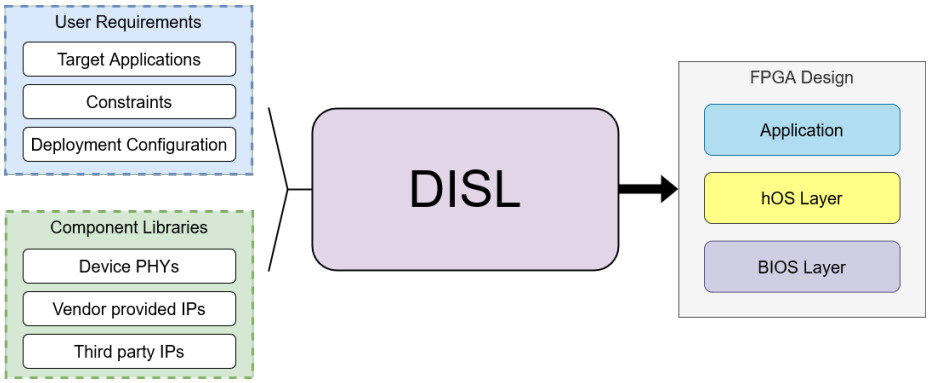
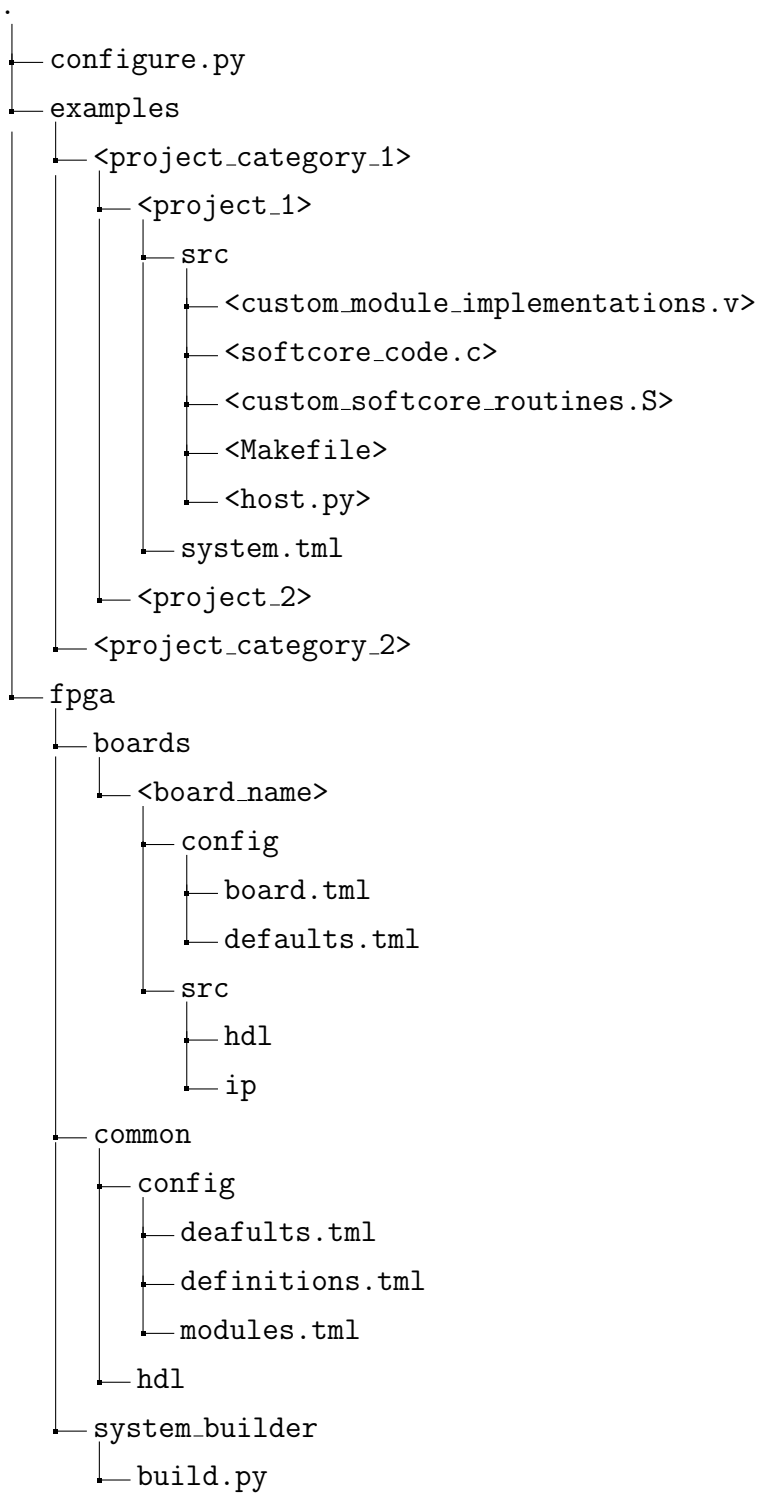


Figure 2·1: Simplified Overview of DISL.

DISL does not depend on any APIs specific to a particular vendor or tool flow. Any device/vendor/tool-specific aspects are part of the components. It is possible for a user to use a component without a full understanding of the device/vendor/tool-specific details. It is the responsibility of the component designer to properly abstract these when adding components to DISL. DISL uses configuration files written in TOML [Preston-Werner, Tom and Gedam, Pradyun et al., 2023] syntax. TOML is a minimal configuration file format with simple semantics. There is existing support in many popular programming languages for parsing TOML files. DISL is implemented using the Python programming language.

2.2.1 Directory Structure

The directory structure of DISL is shown here. The files in this directory structure are referenced in discussions throughout this dissertation.


.
configure.py
examples
<project_category_1>
<project_1>
src
<custom_module_implementations.v>
<softcore_code.c>
<custom_softcore_routines.S>
<Makefile>
<host.py>
system.tml
<project_2>
<project_category_2>
fpga
boards
<board_name>
config
board.tml
defaults.tml
src
hdl
ip
common
config
deafults.tml
definitions.tml
modules.tml
hdl
system_builder
build.py

2.2.2 DISL Terminology

Next, we go over parts of DISL terminology and provide examples for some. Since we use DISL to implement all the designs used in this dissertation, our later discussions use similar terminology to DISL.

System Definitions

DISL uses a set of system definitions to establish the common nomenclature used in all the configuration files. DISL system definitions are given in `fpga/common/config/definitions.tml`. The system definitions include:

Namespaces: DISL uses namespaces similar to software. They help resolve any conflicts between signals and interfaces from different parts of the design but with the same name. Eg: `'MODULE:ddr'`, and `'BOARD:ddr'`. Here, `MODULE` and `BOARD` are namespaces.

Interconnects: Defines the types of interconnects DISL can generate. Eg: `'ONE_TO_ONE'`, `'ONE_TO_MANY'`.

Intrinsics: These are templates to generate simple hardware constructs as part of the top module. Listing 2.1 shows definitions of two intrinsics for signal assignment and concatenation.

```
[INTRINSICS]
ASSIGNMENT = "wire [%{CUSTOM_SIGNAL_WIDTH} - 1 :0] %{CUSTOM_SIGNAL_NAME};
              assign %{CUSTOM_SIGNAL_NAME} = %{INPUT_SIGNAL}%{SIGNAL_BITS};"
CONCAT2 = "wire [%{CUSTOM_SIGNAL_WIDTH} - 1 :0] %{CUSTOM_SIGNAL_NAME};
           assign %{CUSTOM_SIGNAL_NAME} = {%{SIGNAL_1}, %{SIGNAL_2}};"
```

Listing 2.1: Intrinsic definitions

Handshakes: DISL defines communication protocols in terms of the handshakes the protocol is using to communicate. DISL does not specify a fixed set of supported

communication protocols. Instead, a user can add interface definitions to the system definitions. DISL uses handshake based protocol definitions to describe communication protocols and the interfaces involved in communication. A single handshake comprises a 'READY' part, a 'VALID' part, and an arbitrarily sized 'FRAME' part that is created by concatenating zero or more signals. Listing 2.2 shows the handshakes portion of the system definitions.

```
[HANDSHAKES]
  TYPES = ["READ_ADDRESS", "READ_DATA", "READ_RESPONSE", "WRITE_ADDRESS",
           "WRITE_DATA", "WRITE_RESPONSE", "READ_RESPONSE_ADDRESS","NONE"]
  DIRECTIONS = ["REQUEST", "RESPONSE", "BIDIR"]
```

Listing 2.2: Handshake definitions

Protocol Interfaces: Protocol interface definitions are added to the 'PROTOCOL' dictionary in 'definitions.tml'. Listing 2.3 shows an example protocol definition of an AXI-lite interface. (Not all the handshakes are shown here. The rest of the handshakes follow the same format). Similarly, Listing 2.4 shows the definition for a custom protocol that only uses one handshake.

```
[PROTOCOLS.AXIL]
  [PROTOCOLS.AXIL.WIDTHS]
    "wdata"    = "DATA_WIDTH"
    "wstrb"    = "MASK_WIDTH"
    "wvalid"   = 1
    "ready"    = 1
    "data"     = "DATA_WIDTH"
    "rvalid"   = 1
    "ready"    = 1
    "..."      = "...    "
  [PROTOCOLS.AXIL.HANDSHAKES]
    [PROTOCOLS.AXIL.HANDSHAKES.WRITE_DATA]
      DIRECTION = "REQUEST"
      READY = "ready"
      VALID = "wvalid"
      FRAME = ["wdata", "wstrb"]
    [PROTOCOLS.AXIL.HANDSHAKES.READ_DATA]
      DIRECTION = "RESPONSE"
      READY = "ready"
      VALID = "rvalid"
      FRAME = ["rdata"]
    [PROTOCOLS.AXIL.HANDSHAKES. ...]
```

Listing 2.3: Example protocol definition (AXI-lite)

```
[PROTOCOLS.DESC_BYPASS]
 [PROTOCOLS.DESC_BYPASS.WIDTHS]
  "ready_0" = 1
  "src_addr_0" = "DATA_WIDTH"
  "dst_addr_0" = "DATA_WIDTH"
  "len_0" = 28
  "ctl_0" = 16
  "load_0" = 1
 [PROTOCOLS.DESC_BYPASS.HANDSHAKES]
  [PROTOCOLS.DESC_BYPASS.HANDSHAKES.WRITE_DATA]
   DIRECTION = "REQUEST"
   READY = "ready_0"
   VALID = "load_0"
   FRAME = ["src_addr_0", "dst_addr_0", "len_0", "ctl_0"]
```

Listing 2.4: Protocol definition (Custom protocol)

Component Library

The component library includes discrete components used by the DISL framework to generate systems consisting of application logic and infrastructure logic to provide services to the application logic. The components range from device-specific blocks (including definitions for PHYs), to completely generic IP blocks. Users can create component libraries that are customized to their specific domain. The steps to add new components to the component library are described in Section 3.3.1.

System Configuration

The minimum requirements for a DISL project directory include the project-specific dependencies folder `'src'` and a TOML file named `'system.tml'` that contains the system configuration. The system configuration file is the primary interface to DISL for generating systems. It contains all the information required by DISL to instantiate, parameterize, and connect the necessary modules.

Naming Conventions: DISL uses namespaces to reference signals and parameter values in the configuration file. References are made using a set of substrings joined by the colon character. The first substring is always a namespace as defined in

‘fpga/common/config/definitions.tml’.

MODULE:<instance_name> refers to an instantiation of a module. MODULE:<instance_name>:<interface_name> refers to an interface of this instantiation. To refer to a signal in this interface, :<signal_name> is appended to the previous string. BOARD:<io_name> is used when referring to board I/O. Similarly, when referring to a custom signal generated using intrinsics, CUSTOM:<signal_name> is used.

External I/O: A list of required I/O should be provided as the PORTS parameter in the EXTERNAL_IO dictionary.

Instantiations: The INSTANTIATIONS dictionary is used to specify: the modules to be instantiated, parameter overrides, and any additional parameters that can be used to guide the system builder. Listing 2.5 an excerpt from a system configuration file including the INSTANTIATIONS dictionary.

```
[INSTANTIATIONS]
    [INSTANTIATIONS.cache]
     MODULE = "bram"
     PARAMETERS.MEMORY_SIZE = 256
    [INSTANTIATIONS.timer]
     MODULE = "timer_axi"
     PARAMETERS.CLOCK_FREQ_MHZ = 12
    [INSTANTIATIONS.debug]
     MODULE = "uart_axi"
     PARAMETERS.CLOCK_FREQ_MHZ = 12
     PARAMETERS.UART_BAUD_RATE_BPS = 921600
     PARAMETERS.DATA_WIDTH = 32
```

Listing 2.5: Example INSTANTIATIONS

Intrinsics: Intrinsics are specified in the INTRINSICS dictionary as part of the system configuration. Listing 2.6 shows an example of intrinsics in the system configuration file. The first intrinsic is generating a signal named jtag_reset and assigning bit zero of the chip_manager module's signal named control. The second entry assigns

the value `(custom_reprogram XOR 1'b1)` to the custom signal `cpu_resetn`.

```
[INTRINSICS]
[[INTRINSICS.ASSIGNMENT]]
  CUSTOM_SIGNAL_WIDTH = 1
  CUSTOM_SIGNAL_NAME = "CUSTOM:jtag_reset"
  INPUT_SIGNAL =  "MODULE:chip_manager:control"
  SIGNAL_BITS = "[0]"
[[INTRINSICS.COMBINATIONAL]]
  CUSTOM_SIGNAL_WIDTH = 1
  CUSTOM_SIGNAL_NAME = "CUSTOM:cpu_resetn"
  INPUT_SIGNAL_1 =  "1'd1"
  OPERATION = "^"
  INPUT_SIGNAL_2 =  "CUSTOM:reprogram"
```

Listing 2.6: Example Intrinsics Usage

Interconnect: The system interconnect is specified using three separate data structures in the 'INTERCONNECT' dictionary. Two out of the three structures (Static and Dynamic) can only connect interfaces while the last one (Override) can only connect individual signals of interfaces (or connect single signal interfaces). Listing 2.7 presents the template for the INTERCONNECT dictionary.

```
  [INTERCONNECT]
      OVERRIDES =  [["..."], ["..."] , []]
      STATIC = [["..."], ["..."] , []]
      [INTERCONNECT.DYNAMIC.<module_name_1>]
      [INTERCONNECT.DYNAMIC.<module_name_2>]
```

Listing 2.7: INTERCONNECT Dictionary Template

OVERRIDES: Overrides are used to assign values to individual signals ignoring all other assignments in case the target signal is part of an interface for which the connectivity is specified in one of the other two components of the interconnect dictionary.

STATIC: This list within the INTERCONNECT dictionary specifies the connections that do not change over time, and are between interfaces of the same type.

DYNAMIC: Any connection that does not fall in the previous two lists is added to the DYNAMIC list. These represent complex connectivity among modules that require some form of arbitration/selection logic, or scenarios where protocol conversion is necessary.

System Builder

The DISL system builder translates the information from the configuration files described previously into a valid system build. The system builder is defined in the `'BUILD'` class in `fpga/system_builder/build.py`. A `BUILD` object is instantiated with three arguments:

1. System specification – Location of the `system.tml` file,
2. Name of the target board, and
3. build directory

All other configuration files are loaded automatically.

The system builder generates the RTL for the `top` module and the Verilog header file containing the parameter values using the system specification in `system.tml`, definitions of modules and interfaces in `modules.tml` and `definitions.tml`, and the information regarding board I/O ports provided in `board.tml`. The design constraints are copied from `board.tml` to a constraints file (eg: `constraints.xdc`). Commands to generate/modify IP cores are similarly copied to `ip.tcl`. Finally, the system builder copies all the necessary source files to the build directory based on the dependencies specified in `modules.tml` and `board.tml`.

2.3 Host-FPGA Communication

2.3.1 Background

Typical Use Model for PCIe FPGAs

Most high-end FPGAs, and even some low-cost FPGAs such as [Xilinx, 2022a], currently support PCIe connectivity. This makes PCIe a popular host-device communication mechanism for FPGAs mainly because of high data transfer speeds making it ideal for high-performance computing tasks that require lots of bandwidth. The

host-FPGA communication is typically carried out with the assistance of a device driver either provided by the FPGA vendor or written by the user. In either case, the driver is specific to the given device. This is unavoidable because of different capabilities and PCIe IP core availability across different FPGA families. This results in the user having to maintain different drivers for different device families, and even for the same device, depending on the PCIe IPs used. There are open-source PCIe IPs and drivers provided by third parties such as [XILLYBUS, 2022], and [Richmond et al., 2022]. These have similar limitations such as being specific to a device or possibly a particular board, and the user having to rely on a third party to maintain the drivers.

More recently, FPGA vendors have been providing runtime libraries, such as Xilinx runtime library (XRT) [Xilinx Inc., 2022] and Open Programmable Acceleration Engine (OPAE) [Intel, 2017], which provide the user with simple APIs for programming, data movement, and controlling the FPGA. These typically provide a user-space library, kernel space device drivers, and an FPGA shell that takes up a fixed portion of the FPGA fabric to implement the communication infrastructure such as PCIe and Ethernet controllers, DMA engines, memory interfaces, and other peripherals. The kernel drivers are designed to match the IP blocks used to build this shell. The user application kernel is typically instantiated inside the shell by using partial reconfiguration. Some of these runtime libraries are even made open source. However, the drivers themselves are still device-specific and these frameworks do not support all the FPGA devices, even from the same vendor.

There exists a requirement for a more general use model for communication with FPGAs over PCIe. Ideally, the drivers should be agnostic of the device being used, and the user should not have to maintain the device drivers by updating them whenever the kernel is updated. A possible solution that satisfies these requirements is the use of the Virtio drivers that are part of the standard Linux distributions.

Virtio

Virtio devices are virtual devices found in virtual environments, yet, by design, they look like physical devices to a guest within a virtual machine [Tsirkin and Huck, 2022]. Virtio devices use normal bus mechanisms for device discovery, interrupts, and DMA. Virtio drivers follow suit. This allows using Virtio drivers to communicate with physical devices as well. The device in question only needs to present a Virtio-compliant interface to the driver and the driver is agnostic to the fact that it is communicating with a physical device. The device types currently supported in the Virtio specification are shown in Table 2.1.

The Virtio architecture consists of three key components; front-end drivers, back-end devices, and the queues used for all communication between the front- and back-end components called 'virtqueues'. Virtio drivers are the front-end component used by guest applications running in a virtual machine. These communicate with the back-end Virtio devices that are emulated by the host machine. There is a multitude of different combinations for the placement of these components in guest and host user versus kernel space. A detailed description is available in [Pérez Martín, 2020].

An interesting use case is para-virtualization in which there is a physical device attached to the host machine and access to it is virtualized by the Virtio layer. Figure 2·2 depicts an abstract view of this use model. The requests from the Virtio back-end device are sent to the physical device via the legacy device driver running in the host kernel space, and possibly a bridge device which converts the transactions to a format understood by the legacy device driver.

If the physical device exposes a Virtio-compliant interface, the intermediate layers can be bypassed to improve performance. In this scenario, a Virtio driver running on the guest kernel space and one running in the host kernel space see the same interface presented by the physical device. Figure 2·3 illustrates two use models that

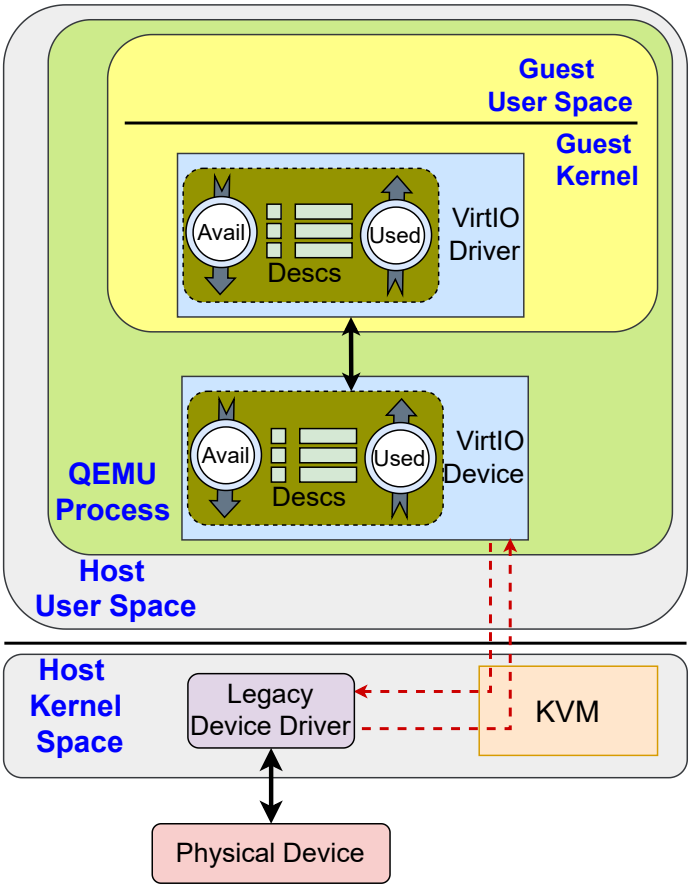


Figure 2·2: Virtio para-virtualization.

such a Virtio-compliant physical device will enable. In Figure 2·3(A), the user space application interacts with the Virtio driver running in the guest kernel space. The physical PCIe device is exposed to the guest VM with PCIe passthrough, which allows the Virtio driver to directly communicate with the physical device. In Figure 2·3(B), the Virtio driver is running in the host kernel space and communicating with the physical device. The guest application directly accesses the host Virtio driver when it needs to use the physical device.

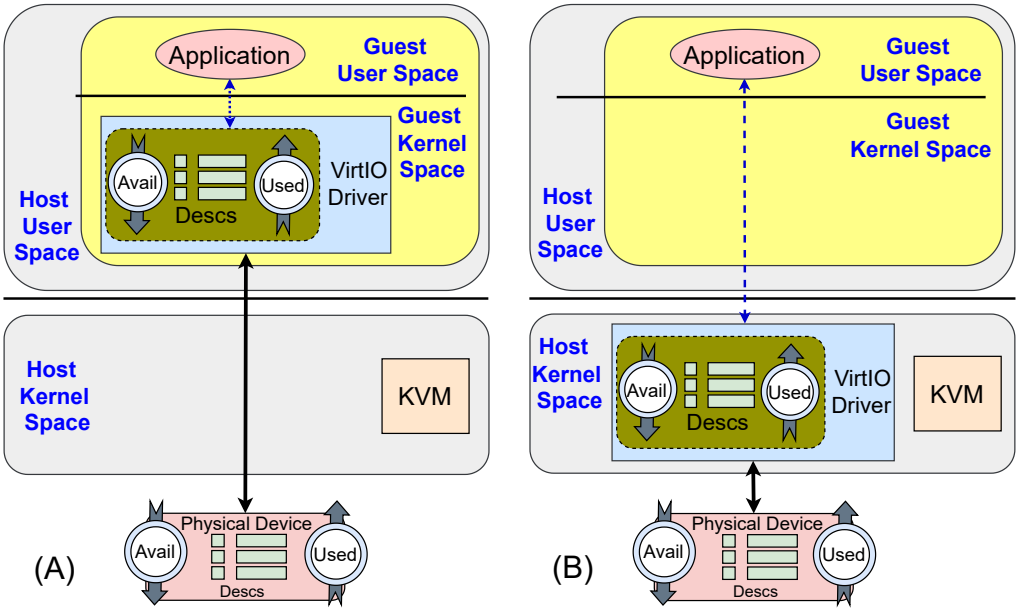


Figure 2·3: New use models for a physical Virtio device.

2.3.2 Related Work

The authors of [Mbongue et al., 2021] 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 [Mbongue et al., 2018]. 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 [Intel, 2023a]. 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 [Silicom, 2021] is based on an Intel Stratix 10 DX FPGA. It can be deployed as a Virtio network or storage accelerator. Details regarding the IP cores used are not available. 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 [RSPwFPGAs, 2020] to implement Virtio support on an FPGA PCIe endpoint. In contrast, the vendor-provided PCIe IP is modified in [Bandara et al., 2022] 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 [Korolija et al., 2020, Heinz et al., 2021, Vaishnav et al., 2020, Agne et al., 2013, Anderson et al., 2006, Zhang et al., 2017, Ma et al., 2020, Fleming and Adler, 2016] focused on enhancing the usability of FPGAs through techniques such as FPGA shells, implementing operating system-like abstractions on FPGAs, automated composition of systems from custom processing elements, FPGA virtualization, and integrating FPGAs into the host operating systems' thread/process abstraction. However, these works do not focus on the portability aspects of host-FPGA communication and depend on vendor-provided or custom device drivers for PCIe communication.

2.4 Applications

2.4.1 Overview

While FPGAs are often considered to be specialized processors, they are notable in their numerous and varied application domains [Gokhale and Graham, 2005, Hauck

and DeHon, 2008, Koeher et al., 2008]. An area where FPGAs have long been prominent is in supporting high performance communication, especially when combined with computation as with SmartNICs [Guo et al., 2022b, Guo et al., 2023] and SmartSwitches [Sheng et al., 2015, Sheng et al., 2017, Sheng et al., 2018, Haghi et al., 2023, Haghi et al., 2024]. HPC applications that are communication bound can sometimes take advantage of this capability directly. One example is Molecular Dynamics [VanCourt and Herbordt, 2006, Chiu, 2011, Wu et al., 2021, Bandara et al., 2024b], which has well-known problems with strong scaling [Wu et al., 2022, Wu et al., 2024]. Another is Molecular Docking [VanCourt et al., 2004, Sukhwani and Herbordt, 2008, Sukhwani and Herbordt, 2010], which often combines Molecular Dynamics with operations familiar from signal processing, also an FPGA domain. FPGA's ability to support flexible data types has led to their use in Bioinformatics [Sajjadinasab et al., 2024b] and Machine Learning [Li et al., 2019, Geng et al., 2021]. A final example given here is Multiparty Computation. When Secret Sharing is used, it has been shown that the FPGA's tight coupling of computation and communication can be beneficial [Wolfe et al., 2020, Patel et al., 2020, Patel et al., 2022].

2.4.2 Key-value Stores

Key-value stores (KVS) are non-relational databases that provide the functionality of an associative array by storing data as key-value pairs. KVS are suitable for storing a large variety of structured and unstructured data types. They can handle large amounts of data at high throughput and support quick read and write operations, often in constant time $O(1)$. With the ever-increasing demand on data centers for high bandwidth access to large quantities of data, distributed in-memory key-value stores such as Memcached [Memcached, 2024] and Redis [Redis, 2024] have become integral middleware applications in the current datacenter infrastructure. Many web service providers use KVS [DeCandia et al., 2007, Chang et al., 2008, Xu et al., 2013,

Nishtala et al., 2013] and offer them to the users as a service [Amazon, 2024, Microsoft, 2024a, Alibaba, 2024, Google Cloud, 2024a].

Due to the importance of key-value databases, a large number of prior works have made various attempts at improving their performance. Firstly, there are the software-oriented approaches focused on speeding up in-memory key-value stores on CPUs. There are algorithmic and data structure optimizations [Fan et al., 2013], parallel data access based methods [Lim et al., 2014], and novel methods for managing storage [Ousterhout et al., 2015]. Kernel-bypass to overcome the limitations introduced by the network stack is also a common strategy [Lim et al., 2014, Ousterhout et al., 2015, Ghigoff et al., 2021]. Another set of work is focused on using RDMA to avoid the limitations introduced by the TCP/IP protocol stack [Tang et al., 2017, Jose et al., 2011, Dragojević et al., 2014]. Apart from academic work, research efforts have been made by large cloud operators as well [Nishtala et al., 2013].

A significant number of research efforts have been made to improve KVS performance using FPGAs. Many of these prior efforts offload the KVS completely or partially to FPGAs to achieve higher bandwidth and lower latencies [Choi et al., 2018, Chalamalasetti et al., 2013, Liang et al., 2016, Lavasani et al., 2013, Blott et al., 2013]. Some FPGA-based work also focuses on caching KVS entries using FPGAs [Fukuda et al., 2014]. There are also implementations that are geared towards specific applications such as Blockchains [Sakakibara et al., 2017, Sanka et al., 2021]. However, all of these works above focus on the KVS node instead of the client nodes. The authors of [István et al., 2018] discuss providing multi-tenant services with FPGAs where all the tenants are using the same service.

Memcached

Memcached [Memcached, 2024] is a free and open-source, high-performance, in-memory key-value store. It is typically used as a distributed caching system for

"small chunks of arbitrary data". Memcached provides a number of operators such as `Get`, `Set`, and `Delete` to manipulate the data objects.

Device ID	Virtio Device
0	reserved (invalid)
1	network device
2	block device
3	console
4	entropy source
5	memory ballooning (traditional)
6	ioMemory
7	rpmsg
8	SCSI host
9	9P transport
10	mac80211 wlan
11	rproc serial
12	virtio CAIF
13	memory balloon
16	GPU device
17	Timer/Clock device
18	Input device
19	Socket device
20	Crypto device
21	Signal Distribution Module
22	pstore device
23	IOMMU device
24	Memory device
25	Sound device
26	file system device
27	PMEM device
28	RPMB device
29	mac80211 hwsim wireless simulation device
30	Video encoder device
31	Video decoder device
32	SCMI device
33	NitroSecureModule
34	I2C adapter
35	Watchdog
36	CAN device
38	Parameter Server
39	Audio policy device
40	Bluetooth device
41	GPIO device
42	RDMA device
43	Camera device
44	ISM device
45	SPI master

Table 2.1: Virtio Device Types

Chapter 3

Hardware Operating System Generator and Component Design

In this chapter, we first present our vision for a hardware operating system generator in terms of a formal definition and ideal characteristics/capabilities. It is important to establish these attributes because they guide the component design strategy discussed later. Next, we examine how well DISL, the system generator used in this work, and other related works demonstrate these ideal characteristics. If they do, we also discuss the different strategies used by different system generators to achieve these desired attributes. In Section 3.3, we discuss how the component design strategy impacts the attributes of the hOS generator and the generated designs. Finally, in Section 3.4, we present a set of design guidelines for hOS components.

3.1 What is an hOS Generator?

Before discussing hardware OS generators, let us revisit our definition of a hardware OS for FPGAs. **A hardware operating system is a layer of hardware that implements and manages lower-level interfaces, and provides services to the application logic on the FPGA**. Capabilities beyond the basic functionality specified above are optional and are included in the hOS generation only if necessary for a given use case as specified by a user.

While they make FPGAs more accessible to a larger user base – especially for users who lack expertise in hardware design – fixed FPGA shell implementations restrict

application portability. The shell itself is not portable across devices. Hardware OS generation according to the application requirements can improve application portability and resource usage. We define a hardware OS generator as **a tool/framework capable of mapping application requirements to a system design and generating the said design using a set of components**.

A general-purpose hOS generator capable of targeting different devices and tool flows from different vendors, and generating systems for any one of the various FPGA deployment scenarios can fundamentally change the current FPGA use model where an FPGA developer (re)designs the entire hardware stack when implementing a new application or porting an existing application to a new device. Instead, a developer can focus on designing the application logic, and use an hOS generator to generate the infrastructure required for the application logic to function. With the flexibility to use non-HDL languages to specify the application logic that is not specific to a device, this opens up FPGAs to a much broader community of users while maintaining a higher level of flexibility to allow expert users to perform deep customizations on the generated infrastructure logic.

Apart from the basic functionality of generating an hOS according to a specification, we can enumerate a number of desirable characteristics for an hOS generator and the generated hOSs.

- The hOS generator can generate hOS designs with all the necessary capabilities and only the necessary capabilities as specified by the user.
- The hOS generator is general-purpose. In this context, this means that the hOS generator can
  - generate a wide range of systems in terms of complexity and features
  - generate systems for any application domain and deployment context.

- A user can specify a design without vendor-specific commands, constraints, or constructs.
- Yet, the hOS generator can target different FPGA toolchains.
- A user can port an hOS design targeting one device to another with no or minimal changes to the specification.
- A system can be specified without defining any device-specific details of the lower-level interfaces except for specifying the target device.
- Advanced users can manipulate the lower-level parameters for the different components of the generated hOS and add new components for new functionality and device support.
- New device support can be added incrementally. This means that when porting an application to a new device, adding the components required by the application is sufficient to generate a valid system.
- The generated hOS presents consistent interfaces to the application logic to facilitate application portability.
- Yet, the interfaces are flexible enough to use existing designs with the hOS generator.
- No restrictions are imposed on how the user logic is implemented/optimized.
- The hOS generator can be easily extended to add new functionality.

Note that the above are ideal attributes for a hypothetical hOS generator. An actual hOS generator may only possess a subset of these.

3.2 hOS Generator Design Choices

In this section, we discuss some alternative design choices for hardware operating systems and an hOS generator. We analyze how these design choices may affect the different attributes listed in the previous section. We also explore how practical implementations of hOS generators handle these design choices. We use DISL [Red Hat, 2024] as the primary point of reference in this discussion. We also use other related works such as [Korolija et al., 2020, Heinz et al., 2021, Xilinx, 2023] as comparison points. Out of these, only DISL and TaPaSCo [Heinz et al., 2021] are system generators. The others are fixed hOS implementations. We include them in this discussion because the attributes we discuss are related to both the hOS generator as well as the generated hOS designs.

3.2.1 Minimum Set of Capabilities for the Generated hOS

The first design choice to be made regarding a hardware OS is its functionality. Prior works on implementing similar abstractions on FPGA target a wide range of functionality. These range from simple implementations that only implement the IP cores necessary to access the external interfaces [Xilinx, 2023] to complex hOS designs that provide advanced capabilities such as virtual memory and memory management [Korolija et al., 2020]. We believe that neither of these two extremes is suitable for an hOS design.

The main argument for moving away from a complex implementation is that the more advanced capabilities are rarely useful for typical FPGA designs. This goes against the first ideal hOS generator attribute we have identified, which is generating systems with only the necessary capabilities. Having capabilities rarely used in the default implementation is undesirable as it unnecessarily increases the resource usage for the hOS. Secondly, this approach also causes challenges when adding support for

new devices. With the minimum set of capabilities being complex and extensive, adding new device support requires more time and effort compared to a design with a minimal set of capabilities. Ideally, it should be possible to add device support incrementally.

Prior works that have chosen a complex implementation typically target a specific deployment context in which the advanced capabilities are useful. For instance, [Korolija et al., 2020] focuses primarily on an accelerator use model for the FPGA. The OS-like abstractions are designed to provide capabilities such as sharing an address space with a CPU process and allocating memory to the accelerators. While it is possible to use the FPGA shell implementations targeting an accelerator use model in other deployment contexts, many of the capabilities of the shell will be useless and will waste FPGA resources. All the previous works that modify the host OS/hypervisor also limit themselves to the accelerator use model. While the focus of these works is on the host software stack, they still rely on an FPGA shell implementation to implement external interfaces and connectivity between the host software and the accelerators. These shell implementations also include logic not necessary for any other use case except for the accelerator use model. Referring to our list of ideal attributes, an hOS generator should be general-purpose in the sense that it should be able to generate hOS for any deployment context. Therefore, the hOS generator should choose a more limited set of capabilities as the default configuration of the generated hOS designs. Any advanced capabilities not useful for a majority of designs can be optionally included in an hOS by a user when necessary for a particular use case.

Our argument against a bare minimum implementation that simply connects a set of IP cores together is that this requires the user to have a more detailed understanding of the IP cores and their interfaces. Therefore, apart from implementing the

IP cores to access the external interfaces, an hOS design should include extra logic that abstracts away some of the intricate details of IP cores and external interfaces, and presents simpler interfaces to the user logic. Additionally, it could also include arbitration logic that allows the interfaces to be shared among multiple user applications. Defining the minimum set of capabilities in this fashion can achieve a balance between ease of use for an application developer and resource overhead.

One benefit of automatically generating hardware operating systems rather than providing a fixed implementation is that it is not necessary to specify a fixed minimum set of capabilities. The capabilities of the hOS can be customized to match a particular application. A user is not forced to use the set of capabilities the hOS developer believes to be optimal. Depending on the application requirements and deployment context, a user can use a set of components/capabilities closer to a bare minimum implementation that minimizes the resource overhead, or a set of more advanced capabilities at a higher resource cost.

DISL, the system generator used in this work generates FPGA designs based on system specifications provided by a user. It selects discrete components from a component library and connects those together to match the user requirements. Therefore, DISL does not formally specify a minimum set of capabilities/components for the generated hOS designs. DISL demonstrates the previously identified ideal characteristics of an hOS generator by generating systems exactly as specified by a user without unnecessary components being included. This means that DISL can be considered a general-purpose hOS generator because it can tailor the hOS designs to any target deployment. For instance, consider an application like network-attached storage, or an IoT application where there is no host machine to control and communicate with the FPGA. DISL can generate an hOS design that includes interfaces such as network, DDR, NVMe, and GPIO but no host-FPGA interface. The other

FPGA shell implementations lack this flexibility. While TaPaSCo [Heinz et al., 2021] is also a system generator, it still uses a fixed implementation for the device-specific ('platform') layer. Therefore, a system generated using TaPaSCo will include IP cores and associated logic unnecessary for some applications.

The lack of a fixed minimum set of capabilities also means that adding support for new devices can be done incrementally. An application can be ported to a new device as long as all the system components used by the application are ported to the new device. In Chapter 6, we demonstrate this by porting an application implemented on a Xilinx Artix 7 FPGA to a Xilinx Ultrascale+ device while the only components available in the DISL component library for the Ultrascale+ device are the interfaces used by the application even though the development board with the Ultrasacle+ device includes peripherals not available on the low-end development board using the Artix 7 FPGA.

The set of features provided by an OS-like abstraction also gives us an understanding of the target audience. For instance, the vendor-provided FPGA shells are mainly focused on users with less experience in hardware design. This becomes evident when looking at the default use model for these shells. Usually, the vendor-provided shells are part of a tool flow and not available in unencrypted form for a user to modify. While it is possible to use RTL designs with the FPGA shell, the default use mode for the tool flow is HLS which is geared towards application developers who prefer using software programming languages over HDL. As opposed to this, an FPGA shell like [Xilinx, 2023] is targeting users more experienced with hardware design.

Generating hardware operating systems based on user requirements allows an hOS generator to be useful for users with a wide range of experience in hardware design. Provided the necessary components are available, DISL can generate hardware OS designs ranging from bare-minimum designs such as [Xilinx, 2023] to complex, feature-

rich designs such as [Korolija et al., 2020]. We can infer that DISL has more flexibility in terms of the systems it can generate compared to related work. The use model for DISL can differ based on the user's level of expertise in hardware design. For instance, a user with less experience might rely heavily on example designs and components added by other users, that are part of the DISL repository. Users with an intermediate level of experience could add new components to DISL. Expert users may modify the hOS generator itself. We discuss these different use models further with respect to ease of use in Chapter 6.

3.2.2 Monolithic Vs Modular Design

Software operating systems typically follow a modular design philosophy. They divide functionality specific to a given CPU architecture and generic into separate modules. This allows software operating systems to work on different CPU architectures without the user having to make any modifications to the OS. However, most of the operating system-like offerings for FPGAs do not follow the same design philosophy. While there is a separation between the FPGA shell and application logic, the FPGA shell is typically implemented as a single unit. While one could identify the subcomponents of the design, there is no clear separation between device-specific and generic logic. This severely reduces the portability of the FPGA shell across devices. Both FPGA vendors and third parties provide FPGA shells specific to different devices. Porting a shell to target another device takes significant engineering effort. Even porting application logic between two shells requires non-trivial effort unless the two shells provide the same interfaces.

Our hOS design philosophy is motivated by and aims to replicate, the decoupling of hardware, OS, and application space in CPUs. This alleviates the current need for developers to build the entire hardware stack for each chip or development board. Any non-trivial FPGA design contains both device-specific and device-agnostic logic.

The device-agnostic logic can be further subdivided into application logic and non-application logic that implements the infrastructure necessary for the application to function. By clearly separating these three categories of logic and implementing consistent interfaces between each layer, we can decouple these layers similar to a software OS. The non-application logic (device-specific and agnostic) is similar to the architecture-specific and generic code in the Linux kernel. This layered approach allows different layers to be implemented independently and all device-agnostic logic to be easily ported between devices.

When we look at prior work on introducing OS-like abstractions to FPGAs, many do not provide a clear separation between device-specific and agnostic logic in the FPGA shell. While the shell implementations for different devices may reuse components, there is no formal separation that results in independent layers within the hOS. TaPaSCo [Heinz et al., 2021] is an exception to this where the generated systems consist of three independent layers. The hOS portion of a generated design consists of two layers. The 'platform' layer contains all the IP cores to implement the external interfaces. The 'architecture' layer connects clusters of processing elements (application logic) and implements aggregation logic for data and control signals. The aggregated signals are connected to the 'platform' layer. Due to this layered design, both the application logic and the 'architecture' layer (device-agnostic part of the hOS) can be ported to different devices by only implementing a new platform layer.

Previous publications on Coyote [Korolija et al., 2020] do not describe a layered hOS design. However, the current version of Coyote [Coyote, 2024] uses a layered approach where a design has three independent layers. The only static layer is called the "Shell Management" layer. The two dynamic layers are the "Services" and "Application" layers. These two layers implement a nested dynamic reconfiguration model where both layers can be reconfigured at runtime. The services layer implements the

external interfaces such as DRAM, HBM, and network and accelerators that provide services to the application logic (eg: TCP/IP service, RDMA RoCEv2). The application layer contains multiple regions applications can be scheduled onto.

The authors of [Vaishnav et al., 2020] also discuss a layered approach to designing an FPGA operating system. However, to the best of our knowledge, the FPGA shell used is implemented as a single layer. DISL takes a different approach to creating a separation between device-specific and agnostic components of the hOS which is discussed in detail in Section 3.2.3.

Interfaces Among hOS Layers

An important consideration when using a layered hOS design is the interfaces between different layers. According to our list of ideal attributes for an hOS/hOS generator, each layer of the hOS should present consistent interfaces to the next layer. Consistent interfaces simplify porting device-agnostic infrastructure logic and application logic between devices. This is analogous to software APIs. A software application can the kernel APIs to access different services from the kernel. The same application can be recompiled without modifications to target different architectures that support the same operating system and hence the same kernel APIs.

All fixed FPGA shell implementations from prior works specify fixed interfaces between the FPGA shell and application logic. Since most FPGA shells use partial reconfiguration to configure the dynamic regions with user applications, the fixed interfaces become even more important. Routing between the shell and FPGA regions cannot change when scheduling different applications. All applications to be scheduled to a particular dynamic region are precompiled along with the FPGA shell and the routing is fixed. Therefore, every application uses the same interfaces down to the placement of individual wires within an interface.

FPGA vendor toolchains that work with FPGA shells typically use HLS as the

default flow. The hardware generated by the HLS tool using the high-level language description can easily follow the interfaces available in the FPGA shell. If a user is to use RTL implementations with FPGA shells, the RTL implementations must implement the correct interfaces to be compatible with the FPGA shell. Academic works such as [Korolija et al., 2020] and [Heinz et al., 2021] also specify fixed interfaces that follow popular communication protocols such as AXI [ARM, 2023] between the hOS and application logic.

One limitation caused by the fixed interface specifications is that existing hardware modules cannot be used as they are with these hOS implementations. An FPGA developer has to rework the interfaces of any existing modules before integrating them with an hOS. DISL handles this issue in a unique way. Because DISL does not use fixed implementations of layers and generates the whole design based on a system specification, it does not define any fixed interfaces between components. Connectivity between the components is also specified by the user and generated by the system generation framework. Additionally, DISL also supports protocol conversions. (Details on how DISL achieves this are available in Chapter 2.) This allows existing hardware modules to be used with DISL without reworking their interfaces. At the component level, each component still presents consistent interfaces to the rest of the system. However, the system generation framework provides sufficient flexibility to the interfaces for DISL to demonstrate both the desired attributes related to interfaces from our list of attributes.

At the moment, DISL does not support dynamic reconfiguration. Depending on how dynamic reconfiguration is added to the DISL model, the relaxed approach to interfaces may have to be modified. However, figuring out how to add dynamic reconfiguration to DISL while maintaining highly flexible interfaces is an interesting future research direction.

3.2.3 Generating hOS Layers

Once the design decision is made to follow a layered design that divides the hOS into device-specific and generic logic layers, we face another decision as to how the different layers are generated and presented to a user. The first choice is to generate different layers as independent components and allow the user to integrate those with the application logic. However, because an hOS generator allows hardware operating systems to be generated per application, there is an opportunity to further reduce the effort by the user by generating a full system design instead of just the hOS layers.

Out of the related previous works, TaPaSCo [Heinz et al., 2021] has two layers in the hOS portion of the design. Out of the two layers, the 'platform' layer that includes the device-specific components is not generated per design. A fixed implementation for each supported device is provided. The 'architecture' layer is generated. Different application-specific implementations of the 'architecture' layer are used with the 'platform' layer.

The most recent version of Coyote [Coyote, 2024] also has two layers in the hOS portion of the system. However, the division of tasks between the layers and how the layers are generated is different from TaPaSCo. The "Shell Management" layer is a static region that manages the two other layers, implements host-FPGA communication, and interacts with the host software stack. The "Services" layer is a dynamic region that implements other external interfaces and hence includes IP cores and other device-specific logic. Both these layers are specific to a given device and therefore, must be implemented per supported device. The application layer is also device-specific to a certain extent because it contains one or more dynamic regions. Depending on the resources within the dynamic regions, the number of regions, and their placement, a given implementation of the application layer may be reusable with some other devices. But the more likely scenario is the application layer also being

generated specifically for each supported device.

DISL is a full system generator. It generates system designs that include application logic, device-specific hOS logic, and device-agnostic hOS logic. A user can add user modules also to the DISL component library and use them alongside the components already available in DISL. A system specification is used to describe the components in the system and the connectivity among them. DISL does not generate hOS layers individually. Instead, the system is composed of components from each of the layers selected to match the application requirements. This approach is somewhat similar to Unikernel designs in the software domain where an application is statically linked with selected OS code to achieve better performance and minimize attack surfaces among other benefits. Essentially DISL generates the device-specific and -agnostic infrastructure layers and the application layer for every design out of the components that match the application requirements. However, there is no formal separation between the layers. As the generated hOS is tailored to a particular application, and a user can easily generate an hOS, there is no necessity to reuse the hOS with different designs manually instantiating hOS and application logic to build systems. If another design needs the same or similar hOS design, a user should reuse the system specification for the hOS and not the generated hOS itself.

3.2.4 hOS Components

Generating hardware operating systems out of discrete components allows the generated hOS to be tailored to the application requirements. The components themselves affect many of the hOS attributes we have discussed previously. Firstly, the availability of components determines the types of systems that can be generated. A wide range of components allows hOSs of varying complexity to be generated for different application domains and deployment contexts. This makes the hOS generator general-purpose according to our definition.

When adding an hOS component to the system generator, creating a clear separation between device-specific and generic logic allows the generic portion to be reused with other implementations of the device-specific logic for the same device or different devices. Therefore, this improves both component reuse and portability. A good component design strategy can shield the user from device- and vendor-specific details that are not directly relevant to the user application. This allows a user to specify a system without fully understanding some of the hardware details and makes the hOS generator a useful tool for a wide range of users with varying degrees of hardware design experience. Abstracting the vendor-specific details might allow the hOS generator to target devices and toolchains from different vendors.

None of the previous works we are aware of have the concept of component libraries that consist of modules a generator uses to build systems. DISL uses a component library that can include device-specific components such as definitions of PHYs, vendor-provided or open-source IP cores which include both device-specific and agnostic logic, and completely generic components such as soft processor cores. A user can also add user-defined modules to the component library. Components that match the user specification are used to generate a system. Because the generated systems are customized to match application requirements and DISL does not specify a minimum set of required components, adding support for a new device can be done incrementally. A user who wishes to port an application to a new device can only implement the hOS components necessary for their application. With more such users contributing more components, the component libraries can grow to cover more capabilities. Designing and implementing hOS components is discussed further in Sections 3.3, and 3.4.

3.2.5 System Specification

How a system is specified relates to many of the desired traits of an hOS generator listed previously. While the component libraries provide the components to be used in a design, a description of the design that describes the components used and the connectivity among them is required by the hOS generator. Such a system description is essentially equivalent to the top module of a hardware design. Therefore, one approach to describe the system is to provide an HDL file for the top module of the design. This approach was used in the earlier versions of DISL where the user provides the system specification with an HDL file. However, this approach requires the user to have a good understanding of the interfaces exposed by the hOS components at the level of individual signals. Another drawback is that any parameter changes for any of the components require updating the HDL file which also requires the user to understand different parameters for system components. Finally, this approach necessitates that there is another configuration file to specify any details such as the target device that cannot be captured in an HDL file.

Another approach is to provide the complete system specification in a configuration file and generate the top-level module using the information provided in the system specification. DISL requires a user to describe a system in terms of the components and the connectivity between them in a configuration file. Similar configuration files are used to specify the library components, basic system definitions such as interfaces and interconnects, and information regarding the platform (i.e. the FPGA development board).

According to our list of ideal hOS generator characteristics, the notation/language used in the configuration files should be flexible enough to allow different levels of detail depending on the user's level of expertise. A beginner should be able to provide a minimum level of detail and get an application up and running while an advanced

user should be able to access lower-level parameters of the individual components using some extended notation in the configuration file. DISL uses configuration files writing in TOML [Preston-Werner, Tom and Gedam, Pradyun et al., 2023] syntax. DISL configuration files are very detailed when describing components and specifying systems. A user with a good understanding of the hardware level parameters of different components can manipulate the parameter values directly from the configuration files. Also, after any parameter changes a user can run DISL again to easily regenerate the design with the updated parameter values. A user without a deep understanding of the different components can rely on example designs to get started with DISL and generate a system that matches their requirements.

TaPaSCo [Heinz et al., 2021] also uses configuration scripts to in generating the different layers of the system. These use the Xilinx Vivado Tcl APIs. Therefore, the system generator is tied to a specific vendor's toolflow. TaPaSCo uses different scripts to generate the three levels of abstractions in the generated systems. The first handles generation of processing elements (PE) using HLS tools. The scripts configure the HLS tools to generate hardware modules that implement the correct interfaces used in TaPaSCo. The next set of scripts combines multiple PEs into clusters to generate the 'Architecture' abstraction layer. Finally another set of scripts generate the 'Platform' layer which includes all the device-specific components and connects the architecture layer to the platform layer. Out of these, the platform layer has a fixed set of components depending on the target device. The PE level scripts impose a strict interface definition on the application logic. The architecture abstraction layer maps the PEs into a strict (and somewhat unnecessary) hierarchical cluster architecture.

DISL also only supports Xilinx tools currently. However, there is the possibility of extending it to support other tools. Because the configuration scripts do not

depend on any vendor-specific APIs, they provide an additional layer of abstraction between the FPGA tools and the system description. This allows the hOS generator to potentially generate hOSs and matching compilation scripts, Makefiles, and other tool/vendor-specific sources to target different devices and toolchains using the same system specification.

In the case of DISL, because the component library provides all the components of the system, the major task for the hOS generator in terms of RTL generation is to generate the interconnect to connect all the elements of the system. The interconnect is generated based on the system connectivity described in the configuration file. The user is allowed to specify the connectivity at the interface level or the level of individual signals. There is also the ability to override the connectivity for individual signals of an interface. DISL specifies interfaces in terms of the handshakes involved in communication. As long as the interfaces of the application logic can be described in terms of the same handshakes, connectivity between the application and the hOS can be described at the interface level and generated automatically. In the case of interfaces that do not match the supported handshakes, the connectivity has to be described at the level of individual signals. Because DISL does not specify a fixed set of supported interfaces and instead, allows a user to add new interfaces to the system definitions (in `definitions.tml` file), existing application logic can be used with DISL without overhauling their interfaces to match the interfaces presented by the hOS, which is the predicament a designer faces when using other FPGA shells.

3.2.6 hOS Generator Output

The final design decision we discuss here is the output of the hOS generator. A straightforward and simple approach is for an hOS generator to only generate the hardware design for the hOS and the generator output to be a set of HDL files. However, this leaves the user with the tasks of adding design constraints and setting

up the FPGA tools. By including system constraints such as the target device and target operating frequency in the system specification, the hOS generator can also create a full compilation environment by generating constraint files, project files for FPGA tools, and Makefiles.

Fixed hOS implementations such as [Korolija et al., 2020] can simply include the necessary system constraints and other tool-specific scripts alongside the hOS design itself. While [Heinz et al., 2021] is a system generator, it uses a fixed 'Platform' layer and therefore, most of the system constraints are fixed for a given device. In contrast to these, DISL generates the full system out of discrete components. Therefore, it requires the component designers to include any device and vendor-specific constraints and commands in the configuration files that describe the target development board (`board.tml`). If the chosen device is not already included in DISL, a user has to create a directory for the target development board and create the `board.tml` file with the necessary details. When adding new components to DISL, any device-specific information is added to the same configuration file.

3.3 Component Design and hOS Generator Attributes

The capabilities of a hardware OS generator depend heavily on the components it uses to generate systems. What we refer to as a component here is a hardware module that implements a particular capability within an hOS design. (Ex: PCIe, Network, UART) A component can internally instantiate any number of sub-modules which could be user-defined or IP cores. A component provides a service to either the user application or the other components of the hOS. For example, a PCIe subsystem provides the host-FPGA communication to the user application. A soft processor core within an hOS can be used to execute user code as well as to initialize and manage other hOS components.

Some of the attributes of an hOS generator affected by the components are;

1. Ability to generate systems with all the necessary capabilities and only the necessary capabilities.

2. Generate systems of varying complexity for any application domain/deployment context

3. Shield the inexperienced users from device-/vendor-specific details of components

4. Giving experienced users access to the low-level hardware parameters

5. Target different toolchains

6. Consistent interfaces between hOS layers and application logic

7. Easily porting designs across devices

The component design strategy should focus on supporting as many of these attributes as possible.

For the hOS generator to possess the first attribute, the components should be discrete units that are independent of other components. This allows the hOS generator to add components to a design without having to add other components that are unnecessary to meet the user requirements, but because the first component depends on the second. Even if a component depends on another in certain configurations, there should be an alternative to that in other configurations, For example, FPGA designs with PCIe connectivity can use the clock signal from the PCI slot to generate internal clock signals and drive the rest of the design. Typically, the PCIe IP core has a clock output that matches the frequency of data and control interfaces of the IP. Using this signal to drive the rest of the design simplifies the design as the whole

design is in a single clock domain. Now consider a UART module that uses its input clock signal to ensure that it maintains the correct Baud rate. If the UART module was designed with only the clock signal from the PCIe IP in mind, it may not function properly when a different clock signal with a different frequency is used in a design without a PCIe module. Even if the design included a PCIe module, depending on the number of lanes in the target device and the target data rate, the interfaces of the PCIe IP may operate at different frequencies which means the output clock frequency also could change between different implementations. Therefore, a properly designed UART module will parameterize both the target Baud rate and the input clock frequency. This allows the module to function correctly in different designs under different configurations.

The second attribute is mostly determined by the availability of different components. A wide range of components allows hOSs of varying complexity to be generated for different application domains and deployment contexts. However, an argument could be made for highly parameterized components with versatile interfaces that could be used in a variety of system configurations. For example, the PCIe subsystem described in Chapters 4 and 5 can present one of three different interfaces to the user logic on the FPGA. It can be configured to present AXI memory mapped or AXI streaming interfaces which most hardware designers are familiar with. Additionally, it can also present a Virtqueue interface. (Virtqueues are part of the Virtio specification.) This is useful in a configuration targeting a user more familiar with software development where user code is executed on a soft processor core on the FPGA. The user could write software to interact with the queue data structures in FPGA memory to communicate with the host machine.

Attributes 3 and 4 are related to how a component is represented in an hOS generation framework and the use model for a user to add components to a design.

Additionally, the component design strategy could also have an impact. For example, a component could instantiate one or more IP cores or user modules with many hardware parameters a user may not be familiar with. The component designer can use simpler user-facing parameters for the top module of the hierarchy which the user adds to a design and internally calculate the values for the actual hardware parameters. Going one step further, DISL allows the component designer to add a Python function to the system generator to perform such parameter conversions. If the designer wishes to allow advanced users access to the hardware parameters, those still can be exposed in the top module alongside an additional parameter to decide whether the user-provided or internally calculated versions of the parameters are used when instantiating the sub-modules/IP cores.

The ability to target different toolchains and the ability to easily port components across devices depend on whether the generic and device-/vendor-specific portions of a component are clearly separated. When they are, the generic portion can be reused while the rest is replaced with modules that match a different device/toolchain. When adding the device-specific units to the hOS generator, the component designer could also include scripts with the commands to generate and integrate the IP cores into the rest of the component. Listing 4.1 provides a good example of this. The tool-specific commands to generate and modify the PCIe IP core are added to the component library as part of the component description. This allows the system generator to automatically generate the IP core with no user intervention. When porting the component to a different device (different toolchain as well if the new device is from a different vendor) the device-specific portion can be replaced without changing the functionality or the interfaces of the user applications.

Apart from enabling and enhancing the desired characteristics of hOS generators, a good component design strategy can compensate for certain limitations of hOS gen-

erators. For example, consider conditional module instantiation. This is a standard feature in hardware description languages such as Verilog and VHDL. In Verilog, this can be achieved either with `‘generate if’` statements or compiler directives which are similar to C language preprocessor directives. The method/syntax used by a system generator to describe a system may lack the capability to describe this behavior. However, the component design can compensate for this limitation. A wrapper module can be provided to include the different components that need to be instantiated conditionally and provide a parameter to the hOS generator to set when generating the top module of the design and select which submodule is instantiated. The two versions of the PCIe subsystem shown in Figure 5·8 is an example of such a scenario. The top module for the PCIe subsystem can provide a parameter for the hOS generator to specify the target device. Depending on the value of the parameter, either the `‘xdma_xc7’` or the `‘xdma_cvp13’` modules can be instantiated internally. This parameter can even be hidden from the user because the hOS generator can set the parameter based on the information already provided by the user in specifying the target device for the design.

Another example is specifying multiple instances of the same module with a loop construct. In Verilog, this can be achieved with `‘generate for’` statements. If the hOS generator does not allow specifying module instances in this fashion, the component design can again provide a workaround. A wrapper module can be provided with a parameter to specify the number of instances to be created. The input and output ports of the wrapper will have the concatenated I/O of the internal module instances.

3.3.1 DISL Component Design Steps

In the next chapter, we implement a PCIe subsystem and add it the the DISL component library. As a precursor to that, we revisit the steps a user should follow

when adding a new component to DISL. The four main steps involved in adding a component to the DISL component library are;

1. Pre-Processing
2. Adding board support
3. Adding the generic logic to the library
4. Adding parameter mapping function

Pre-processing

During this step, a component is divided into device-specific and generic blocks. This involves making two code partitions `'Board'` and `'Generic'`. `Generic` contains all the logic that is agnostic of the exact device used and hence can be ported to different devices without any modifications. Meanwhile, `Board` includes all the device-specific elements, parameters, and functionality specific to a target device/board (e.g. PHYs, external I/O bus widths). Whenever a component needs to be ported to a different device, this portion needs to be reimplemented to match the new device. Each of the code partitions can be further split into two logical partitions `'HDL'` and `'Parameter'`. Figure 3·1 provides a visual representation of the code partitioning scheme.

The `HDL` partition refers to the file(s) that contain(s) the hardware block's source code. Although the logical partition is named `'HDL'`, the source files are not limited to hardware description languages. It could also include scripts to generate (and modify if necessary) IP cores. This is common in the `Board` partition because many external interfaces require using vendor-provided IP cores and/or PHYs. When the `HDL` logical partition includes a script to generate the source instead of the actual source, a dummy file that acts as the top-level source file is also added alongside the

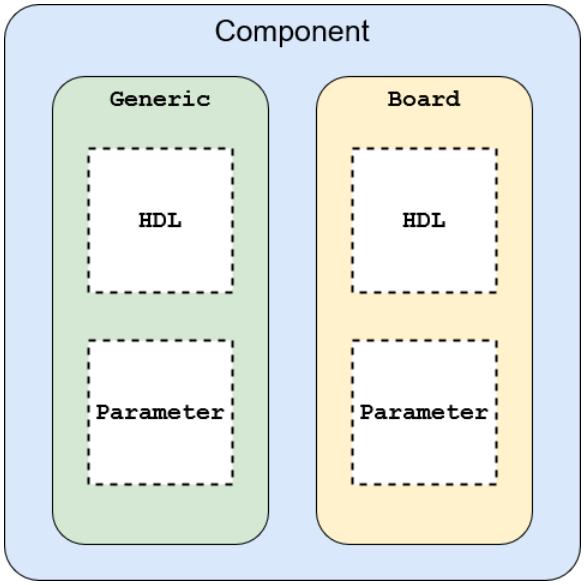


Figure 3·1: Code Partitioning Scheme.

script. When the actual source is generated at build time, this file is replaced. This is necessary because DISL handles dependencies per source file. In the configuration files, a user should specify dependencies with this dummy top-level file. When DISL generates an hOS design, all dependencies are copied over to the build directory.

The `Parameter` partition refers to the top-level parameters of a block and the default values of these parameters. For each hardware block (i.e. per code partition), it is also possible to create an abstraction function in Python that can map a user-facing set of parameters to the actual block parameters. This can simplify the process of customizing the block since it can be non-trivial to correctly identify and change hardware parameters to achieve a high-level goal. Moreover, the Python function can also be used to generate non-parameter data such as initialization files and scripts. Finally, it should be noted that a given component doesn't need to have all the code and logical partitions. A component with no device-specific elements can omit the `Board` partition while a fully device-specific block can omit the `Generic` partition.

The motivation behind dividing a component into `Board` and `Generic` partitions

is improving portability across devices. Here DISL is leveraging the fact that most FPGA designs include both device-specific and agnostic logic and the device-agnostic logic can be used on different devices with no or minimal changes. Deciding the boundary between device-specific and generic logic, however, is not straightforward and is discussed further in Section 3.4.

Board Support

When adding a new component to the DISL component library, if the target device is not already supported, a user is required to add a description of the device to DISL in the form of a configuration file. Here, we briefly cover the steps to add board support for a new target development board.

DISL depends on the information provided in `'fpga/boards/<board_name>/config/board.tml'` to implement the device-specific portion of the hOS. The contents of `board.tml` include descriptions of the FPGA board and board-specific hardware block partitions. The description of a board could consist of information such as the FPGA vendor details, FPGA part number, external pins, their direction (source/sink), and pin location constraints for the external pins. Additionally, commands to generate/modify IP cores, that are part of the `Board` partition are also added to the same file. Listing 3.1 shows excerpts from the `board.tml` file describing an FPGA development board based on a Xilinx Artix-7 xc7a200tfbg484-2 device. This is the target device for our initial implementation of the PCIe subsystem.

```
1  [DESCRIPTION]
2  NAME = "Alinx_AX7A200T"
3  DIRECTORY = "alinx_ax7a200t"
4  CHIP_VENDOR = "Xilinx"
5  BOARD_VENDOR = "Alinx"
6  VENDOR.DDR = "Micron"
7  FAMILY.SHORT = "xc7"
8  FAMILY.LONG = "7series"
9  PART.SHORT = "xc7a200t_0"
10 PART.LONG = "xc7a200tfbg484-2"
```

```
...
[REQUIREMENTS]
[REQUIREMENTS.FILES]
[REQUIREMENTS.FILES."ddr_phy.v"]
  HDL =[]
  IP = ["ip_mig_params_0.prj"]
[REQUIREMENTS.FILES."xdma_xc7.sv"]
  HDL = ["xdma_0_axi_stream_intf.sv", "xdma_0_pcie2_ip_core_top.v", "
    xdma_0_rx_demux.sv", "xdma_0_tgt_cpl.sv", "xdma_0_tgt_req.sv"]
  IP = []
...
[REQUIREMENTS.IP]
[REQUIREMENTS.IP."xdma_xc7.sv"]
xdma_0 = """
    config_ip_cache -disable_cache
    create_ip -name xdma -vendor xilinx.com -library ip -version 4.1 -
    module_name xdma_0
    ...
    <TCL commands>
    ...
    """
...
[IO]
  [IO.sys_clk_p]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "CLOCK"
  [IO.sys_clk_n]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "CLOCK"
  [IO.sys_rst_n]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "GENERAL"
  [IO.pci_exp_txp]
    DIRECTION = "SINK"
    WIDTH = 4
    INTERFACE_TYPE = "GENERAL"
...
[CONSTRAINTS]
sys_clk_p = """
set_property LOC IBUFDS_GTE2_X0Y3 [get_cells refclk_ibuf]
create_clock -period 10.000 -name sys_clk [get_ports {sys_clk_p}]
"""
...
set_property CONFIG_MODE SPIx4 [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 50 [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
...
```

Listing 3.1: Board description example

In Listing 3.1, lines 1 through 10 describe the FPGA development board including part numbers and device vendors. The requirements section (line 13) specifies the files and IP cores required. Notice that the dependencies are specified in relation to source files `'ddr_phy.v'` (line 15), and `'xdma_xc7.sv'` (line 24). These are the source files for the top-level modules of the `Board` partitions for the DDR and PCIe subsystems. The dependencies themselves can be IP cores or other source files. Line 17 specifies a script with the TCL commands to generate an IP core required by `ddr_phy.v`. Line 19 specifies multiple HDL files required by `'xdma_xc7.sv'`. From line 25 onwards, there are several TCL commands used to generate the vendor-provided PCIe IP core (`xdma_0`). The `[IO]` dictionary specifies all the input/output interfaces, their directions, and widths (lines 34-50). Finally, the `[CONSTRAINTS]` dictionary specifies the different constraints including ones related to the development board, I/O pins, and clocks.

Adding support for a particular device is a one-off task. Therefore, once a developer with the necessary expertise creates a board specification, any user can use it without possessing the expertise to create such a specification. Notice that some of the commands for generating IP cores, specifying constraints, and placing device-specific primitives (eg: IBUFDS_GTE2_X0Y3 on line 55) could be vendor- and tool-specific. The board specification file acts as an abstraction layer between the device-agnostic logic and such vendor- and tool-specific elements. This simplifies the process of porting a hardware OS design to different devices.

`Generic` Logic

After adding the device-specific elements to DISL as described in the previous section, a user is left with source files containing device-agnostic logic only. These are added to the `'/fpga/common/hdl'` directory and the topmost module in the hierarchy is described in the `'/fpga/common/config/modules.tml'` file. The rest of the source

files are specified as `'[REQUIREMENTS]'` for the top module using a syntax similar to what is used in `'board.tml'`.

Parameter Mapping Function

As the final step in adding a component to DISL, an optional Python function can be defined to map user-facing abstracted parameters to actual HDL parameters. This is done by adding the function to `'fpga/system_builder/build.py'`. In addition to mapping parameters, this function can also be used to generate component-specific files. For example, if using a softcore processor, this function can be used to generate the linker scripts and reset handlers using the memory maps and the memory sizes in the system specification.

This is an optional step that may not be necessary for every component added to the DISL component library. This step is useful when the component added has complex configuration parameters that are too complex to be exposed to a user with little hardware design experience. A component designer can create an abstraction layer between the user and the actual hardware parameters by specifying simpler user-facing parameters and using a parameter mapping function to derive the component parameters from the user-facing parameters.

3.4 Component Design Strategy

In this section, we combine our discussion from previous sections and the insights gained from implementing a PCIe subsystem for DISL (described in Chapters 4, and 5) to develop a set of design guidelines to follow when designing components for hOS generators.

Our design guidelines are as follows.

- Implement components as discrete units independent of other components in

the hOS. If it is necessary to depend on inputs from other components, include extra logic and parameters that account for different variants of the connected components (including the case where some components are not included in the design).

- Include different variants of the interfaces that may be useful when using the component in different system configurations. Add a configuration method for the hOS generator to select the correct interface variant based on the user requirements.

- If the component has complex hardware parameters, abstract them with simplified user-facing parameters. Add parameter mapping functions either as a part of the component or as external functions depending on the support available in the hOS generator.

- Expose the complex hardware parameters alongside the simplified ones. Provide additional parameters to control which set of parameters are used.

- Clearly separate the device-specific and generic logic within the component.

- Implement consistent interfaces within the component (between the device-specific and generic sub-components) and also between the component and the rest of the system regardless of the exact implementation details of individual sub-components.

- If necessary, implement additional wrapper modules and parameters to account for limitations of the method/syntax used by the hOS generator to describe systems.

Factors Deciding the Boundary Between `Generic` and `Board` Partitions

When implementing an hOS component and porting it to a different device in the

previous sections, we have observed that the separation between generic and device-specific logic is the most important and impactful out of our set of design guidelines. This is also the guideline that will result in the highest variance in terms of the implementation effort and the resultant benefit because there is a multitude of ways a component can be partitioned.

When adding a component to the DISL component library, the user is responsible for partitioning the different elements as discussed above. The user has to decide where the boundary between device-specific and generic logic lies. The effort necessary to create the code partitions relies heavily on this decision. The partitioning should not be done just for the sake of following the recommended steps. The effort to partition the component should not outweigh the benefit of partitioning. Therefore, there are several factors that decide where the boundary is drawn.

- What is the component/interface? Does it have device-specific/-agnostic logic?
  The degenerate case is where a component is fully device-specific or agnostic. In this case, there is no decision to be made as the whole component should go in either the `Board` or `Generic` partition.

- Is the component likely to be reused?
  If a component is unlikely to be reused for other designs, it is difficult to justify the overhead of creating code partitions. In such a scenario adding the component as a single unit without creating code partitions is acceptable.

- Will the component be ported to different devices?
  If a component is not intended to be ported to different devices or if the porting can be done by replacing a subcomponent that includes both device-specific and generic logic, it may not be necessary to perform strict code partitioning. For instance, the PCIe subsystem discussed in the previous sections uses a vendor-provided PCIe IP core as a subcomponent. The IP core itself includes both

device-specific and agnostic logic. However, because the complete IP core will be replaced when porting to a different device family, the source code of the IP core is not partitioned. Instead the whole IP core is added to the `Board` partition.

- Access to IP source files.

  There could be cases where a user does not have access to the source files in unencrypted form for some IP cores provided by either the FPGA vendor or a third party. In this case, the user is forced to include the IP core as a single unit without partitioning.

- What are the interfaces that need to be implemented between the partitions?

  Code partitioning also involves implementing the necessary interfaces between the `Board` and `Generic` partitions. The complexity of the resulting interfaces and the effort necessary to implement the interfaces should also be considered when performing code partitioning.

- Additional constraints.

  A user may have additional constraints such as not using any vendor-provided IPs, performance targets that cannot be achieved using those, or functionality not provided by the vendor-provided IP cores. In such situations, a user might replace all or most parts of IP cores except for hardened ASIC blocks. Then the ASIC blocks will be the only device-specific elements in the design.

- How much time and effort the user is willing to spend?

  Ultimately, the code partitioning decision depends on the amount of time and effort a user is willing to spend. If the initial overhead of code partitioning cannot be amortized by reusing the component, a user may decide to avoid this step and add the component as a single unit. In the case of DISL, the system

generation functionality does not change based on whether the code partitioning is performed or not. Therefore, a user can trade off reusability in favor of faster implementation.

Based on these factors, the partitioning scheme that fits a majority of use cases is what we followed in implementing the PCIe subsystem. There we included the following in the `Board` partition.

1. The IP core
2. Any modules connected to the IP core to extend its functionality (such as the `'xdma_ext_cfg_space'` module in the Ultrascale+ variant of the PCIe subsystem)
3. Wrapper modules that ensure consistency of interfaces between the `Board` and `Generic partitions`

Therefore, we recommend this to be the default partitioning scheme. However, as mentioned above, there are specific situations where different partitioning schemes should be followed.

3.5 Summary

In this chapter, we identified a list of attributes we expect an ideal hOS generator to possess. Then we discussed a set of important design decisions to make when designing an hOS generator. We explored how the system generator used in this work (DISL) and related previous works handle these design decisions and the impact of these decisions on the attributes of the hOSs and the hOS generators. None of the prior works or DISL possess all of the characteristics we have identified. The fixed hOS designs provide the most features but lack the flexibility to select the features

to match the application requirements. System generators like DISL and TaPaSCo provide different levels of flexibility but do not provide the more advanced features available with hOS designs such as Coyote.

Out of the system generator designs, the strategy used by DISL, to generate systems using user requirements and a component library, results in an hOS generator and hOS designs that demonstrate most of the desired attributes we have identified. Future research efforts should focus on adding advanced capabilities such as dynamic reconfiguration to DISL while maintaining the level of flexibility it provides.

We also discussed how most of the ideal hOS generator attributes depend on the components used by the hOS generator and presented a set of design guidelines for hOS components.

Chapter 4

PCIe Subsystem Design: Enabling Virtio Driver Support on FPGAs

In this chapter, we go over a concrete design example implementing a PCIe subsystem for host-FPGA communication, adding it to the DISL component library, and using it in a design. The implementation presented in this Chapter enables the repurposing of Virtio device drivers that are part of standard Linux distributions as generic device drivers for FPGAs. This allows a user to use unmodified Virtio device drivers to communicate with the FPGA without writing their own device drivers or relying on vendor-provided device drivers which are often poorly maintained.

Host-FPGA connectivity is critical for enabling a vast number of FPGA use cases in data centers, edge, and IoT. This interface must be reliable, robust, and uniform, whilst supporting necessary protocols and functionality. However, existing support for host-FPGA connectivity has several drawbacks, both on the host and the device. This includes a lack of portability and poor upstream support, both of which can make it difficult for CPUs to easily and effectively leverage FPGAs. Native Virtio drivers in the host operating system can help address some of these limitations, especially on the host side. However, implementing device-side support for the Virtio specification is a challenge due to the substantial hardware complexity involved.

The framework presented in this chapter enables FPGAs to interface with native Virtio drivers in the host operating system. To reduce the implementation overhead and improve portability, this framework uses both generic RTL blocks and modified

chip/device-specific PCIe IP blocks. Moreover, this approach implements all the necessary data structures and functionality needed to meet the Virtio specification requirements. We test the framework using the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) IP [Xilinx, 2022b], implemented on an Alinx AX7A200 FPGA board (with a Xilinx Artix7 [Xilinx, 2022a] XC7A200TFBG484-2 FPGA chip), and a host machine running the Fedora 37 operating system. Our results show that the FPGA can be successfully enumerated as a Virtio device, and interfaced using only native Linux Virtio drivers.

4.1 Introduction

The flexibility of FPGAs enables them to be important complexity offload devices for the CPU. They can be used to accelerate user and system applications, implement networking functions to process data at line rates, perform system administration, provide secure enclaves with hardware isolation, and a number of other tasks [Caulfield et al., 2016, Xiong et al., 2019, Lant et al., 2020, Haghi et al., 2020, Krishnan et al., 2021, Shahzad et al., 2021, Zink et al., 2021, Haghi et al., 2022, Guo et al., 2022a]. In order to support this vast number of use cases, the host-FPGA connectivity must implement a required set of features and protocols, as well as provide a reliable, robust, and uniform interface.

Despite the critical nature of host-FPGA connectivity, there are a number of limitations of existing PCIe [PCI SIG Org., 2010] interfaces that support this communication. On the device side, the major limitation is portability. The IP blocks used to implement host interfaces are typically vendor-specific, both in terms of chip and board; and can have inconsistencies in the features they support and the APIs they expose to the user logic. This is due to the implementation complexity of the communication stack, differences in resource types/amounts across chips, and the vir-

tually impossible task of building a completely generic hardware stack due to the use of essential ASIC blocks organized in chip-specific IO Bank structures, e.g., SERDES units.

On the host side, the major limitations are portability and poor maintenance. From a portability perspective, the differences in capabilities/functionality of the device can lead to compatibility issues if the driver attempts to use nonexistent functionality. This is especially important in FPGAs given their flexibility and that there is no standard set of supported features. Moreover, similar to the hardware side, APIs exposed to user applications by vendor-provided device drivers can also be inconsistent. From a maintenance perspective, deprecated software dependencies, especially as the host kernel is updated, and a changing set of features supported by IP blocks, means that the device drivers need to be patched frequently. Unfortunately, there is often poor upstream support: most of the work on drivers is done downstream, i.e., by developers modifying the drivers themselves.

Virtio [Tsirkin and Huck, 2022] is an industry standard for I/O virtualization and is one possible solution to the challenges posed by the use of vendor-provided device drivers. There is native support for Virtio in the host operating systems, such as the Linux kernel, which means that no additional driver needs to be written/maintained, and APIs are relatively 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 to virtualized environments, such as faster guest-device communication. Exposing the FPGA to the host as a Virtio device can reduce data copies and latency through direct communication between the guest user space and the host device driver.

While Virtio can help address limitations of the host-device interfaces on the host side, there are two major challenges involved with implementing native Virtio support

on the device. First, the FPGA hardware stack must meet the Virtio specification, which means that appropriate data structures and state machines must be implemented. This is in contrast to existing approaches to Virtio support, which build a hardware stack that does not fully meet the Virtio specification and thus require custom Virtio drivers to interface with it; these have disadvantages similar to the typical FPGA drivers discussed above. The second challenge is that the hardware side is still chip/device specific and existing IP blocks may not support all required functionality. Building up the entire hardware stack from scratch is also not feasible due to the complexity of implementing the required IP blocks.

In this work, we address the above challenges by developing a framework consistent with the DISL component implementation steps, that allows Virtio support to be added to the FPGA hardware stack by: i) building a subset of the required hardware blocks from scratch using generic RTL, and ii) leveraging existing PCIe IP blocks for chip/device-specific parts of the implementation. To achieve this, we first identify the interface that an FPGA should expose to the host to meet the Virtio specification. Next, we instantiate the vendor IP by specifying appropriate parameter values based on the interface requirements. Then, we modify the IP RTL to add/modify critical functionality that was either not available in the IP or was not exposed to the developer when instantiating the IP. Finally, we add a Virtio controller block which only includes generic RTL, as an abstraction layer between the PCIe IP and the user logic. This controller is responsible for implementing additional requirements of the Virtio specification, such as data structures, arbitration logic, queue support, and other state machines. We test our methodology by implementing a Virtio console device on the FPGA.

In our published work [Bandara et al., 2022], we demonstrated the following:

- Enabling FPGAs to leverage native Virtio drivers in the host operating system

for host-device communication;

- Identifying and implementing the device requirements for Virtio, such as data structures, arbitration logic, queue support, and other state machines;
- Improving portability and reducing implementation complexity of the hardware stack by building required hardware blocks using both generic RTL, and modifying existing PCIe IP blocks to implement chip/device specific blocks; and
- Demonstrating the effectiveness of our approach by modifying the Xilinx XDMA IP to enable enumeration and communication as a Virtio console device.

4.2 Methodology

To enable Virtio drivers to communicate with an FPGA over PCIe, the FPGA should present an interface that meets the requirements of the Virtio specification [Tsirkin and Huck, 2022]. These requirements can be divided into three different operating phases: device identification, device initialization, and data movement. The methods used to implement a Virtio-compliant interface are described below in relation to the Xilinx FPGA and IP cores used in our implementation. The same methods are applicable to most Xilinx IP cores. The same methods can be applied to IP cores from different vendors as long as they provide the functionality necessary to implement the Virtio interface.

4.2.1 Device Identification

A Virtio-enabled FPGA should announce the correct vendor and device IDs at the time of PCI bus enumeration, at which the devices on the PCI bus get identified and initialized by the host. The PCI vendor ID should be `0x1AF4`. The PCI device ID is determined depending on the type of Virtio device implemented on the FPGA.

For instance, an FPGA-based smartNIC can use the device ID `0x1041` which is the device ID for a Virtio network device. This is perhaps the simplest requirement for an FPGA implementation. The IP generation tool flow generally allows the user to configure the PCI vendor and device IDs within the GUI when generating the PCIe IP core. Some Xilinx IPs provide the ability to change the PCI IDs at runtime as well. However, this is unnecessary as we only need the device to be recognized as a Virtio device.

4.2.2 Device Initialization

The Virtio specification specifies five PCI capabilities that must be present in the PCI capability list of a PCIe device for it to be initialized and operate as a Virtio device. These are Common configuration, Notification, ISR status, Device specific configuration, and PCI configuration access.

The first four of the above capabilities inform the Virtio driver where to find the corresponding data structures used in initialization and regular operation of the device. The PCI configuration access capability provides the driver with an alternative access method to the data structures located using the other capabilities. This is used in cases where a legacy BIOS cannot directly access PCIe base address registers (BAR) [PCI SIG Org., 2010]. For legacy Virtio devices, the driver expects the data structures to be in the memory or I/O region corresponding to BAR0. However, for modern Virtio devices, each data structure could be mapped to any of the BARs and the Virtio capabilities in the PCI capability list are essential to locating those.

Adding new capabilities to the PCI capability list turned out to be more challenging compared to setting the correct PCI vendor and device IDs. We used a Xilinx Artix-7 FPGA for our proof of concept implementation. This is one of the cheapest FPGAs with PCI capability. The reasoning behind the selection is that, if the capabilities provided by a cheaper FPGA family are sufficient to successfully implement

an interface compliant with the Virtio specification, the same should be true for more expensive device families. Our assumption is that the integrated blocks and IP cores for the more expensive device families will provide similar or better capabilities than what is available for the device family we used.

At this point, it is worthwhile to provide a brief description of the PCIe integrated block and the IP core used in this implementation. A Xilinx XC7A200TFBG484-2 device was used in this work. This device offers the 7-series integrated block for PCI express found in Xilinx 7-series devices. The Artix FPGA uses the second generation (Gen2) integrated block which has fewer capabilities than the more recent iteration of the integrated block (Gen3) used in higher-end device families such as Virtex. The PCIe IP cores internally instantiate the integrated block and use the transaction (TRN) interface of the block to send and receive PCIe transaction layer packets [PCI SIG Org., 2010] (TLP). We use the Xilinx DMA/Bridge Subsystem for PCIe Express (XDMA) IP core [Xilinx, 2022b] for our experiments. In order to modify the PCI configuration space with new capabilities, we first need to understand how the PCI configuration space accesses by the host are handled by the IP core/integrated block, and the interfaces exposed to change the contents of the PCI configuration space. For the device used in this work, the PCI configuration space is implemented as part of the integrated block itself. Under the default configuration, the integrated block responds to configuration space accesses from the host. The configuration space read and write access TLPs do not leave the integrated block via the TRN interface. We have discovered that the integrated block/IP core combination used in our implementation provides two different mechanisms to change the contents of the PCI configuration space. Since we used a lower-end device, we can expect the higher-end devices and corresponding IPs to provide similar or more flexible interfaces. For instance, the 7-series Gen3 integrated block for PCIe provides a more feature-rich

configuration management interface compared to the interface available in the Gen2 block.

Adding new capabilities to the PCI capability list involves two tasks. The first is to add the capability entries to the configuration space. The second task is forming the capability list by correctly setting the next pointer fields of the capabilities.

Updating configuration space contents

Configuration Management Interface: The first mechanism to change the contents of the PCI configuration space is the Configuration Management Interface. Specifically, the signal group referred to as the "Management Interface Ports" allows user logic to read and write to the PCI configuration space. This interface has two shortcomings that make it unsuitable for our requirements. First, as we discovered through our testing, the integrated block for PCIe does not implement the full PCI configuration space as write-enabled registers. Only specific portions of the configuration space are implemented as writable registers and the rest of the configuration space is read-only. This makes it unsuitable for adding capabilities to the configuration space. Even if the whole configuration space was writable by the user logic, this method is still not suitable because the configuration space modifications have to be done after the user logic has started operations. This means that the new capabilities may not be visible to the host when enumerating the device. This is especially true when using Tandem configuration [Xilinx, 2020]. PCI devices have a strict time limit of around 100ms from reset, under which they should be ready for enumeration. If not the device will not be enumerated. When it takes longer than the above time limit to read a bitstream from flash memory and fully program an FPGA, Xilinx devices use a technique called *Tandem Configuration*, where the bitstream is built in two parts of which the first part contains only the bare necessities for PCI enumeration and the second part contains user logic. We wish the Virtio-enabled FPGAs to behave as

any other PCIe device and to be enumerated at system boot up without requiring an additional PCI bus rescan after boot up.

Configuration Space Access Forwarding: The second method is forwarding the configuration space accesses to the user space over the 'TRN' interface. While the 7-series integrated block for PCIe supports this feature, this is disabled in the XDMA IP core because the IP is not designed to handle configuration space read and write TLPs. Therefore, we have to make changes to the XDMA IP itself. There are two distinct ways the same outcome can be achieved.

1. RTL modification: The first method is to modify the source files for the IP core manually. We have to modify the source file which instantiates the PCIe integrated block. The configuration space access forwarding can be enabled by setting the attribute named EXT_CFG_CAP_PTR to an appropriate double-word address. All configuration space accesses for addresses above the specified address are forwarded to the user logic. Before modifying the RTL, the IS_LOCKED property for the IP should be set. Otherwise, Vivado compilation flow resets any modifications made to the RTL when recompiling the IP core. After setting the locked property, and modifying the source code, the IP is recompiled as an out-of-context module synthesis run. Out-of-context synthesis is where the IP is compiled independently of the rest of the design.

   The XDMA IP used in this work exposes an interface named Dynamic Reconfiguration Port (DRP) that allows the user logic to dynamically control attributes of the PCIe hard block. The EXT_CFG_CAP_PTR attribute can be set using the DRP interface. However, it has the same limitation as the configuration management interface described above as the user logic has to be operational and finish a write operation over the DRP interface before the device enumeration for this method to work correctly.

2. Modifying the XML file describing the IP components: The XDMA IP internally instantiates the 'PCIe integrated block', which includes the transceiver PHY and surrounding logic. We can separately generate this IP with configuration space access forwarding enabled, and add it to the XDMA IP. Xilinx IPs use an XML file to describe their sub-components. The second method is to update this XML file of the XDMA IP core and to force the toolchain to use the sub-IP we previously generated instead of the one generated by the Vivado tool flow as part of the XDMA IP. Similar to the first method, we still have to lock the IP and recompile it as an out-of-context synthesis run after updating the XML file.

   It should be noted that the PCIe IP for the FPGA we used in this work is free and the source files are not encrypted. If the IP is provided by the vendor in an encrypted format, a user may not be able to use the methods described here to implement a Virtio-compliant interface. However, we believe that this work still acts as a confirmation that it is possible to implement an interface fully compliant with the Virtio specification and unmodified Virtio drivers can be used to communicate with FPGAs. There is no limitation in hardware preventing such an implementation except for artificial restrictions imposed by the FPGA vendor through the toolchain and encrypted source files.

   After configuring the integrated block to forward the configuration space accesses to the TRN interface, we need to implement the configuration space registers with the new Virtio capabilities, and the control logic to respond to configuration space read and write TLPs. The TLP header field is used to differentiate the configuration space access TLPs from the other TLP types on the TRN interface. Because the transaction interface is converted to an AXI interface by the XDMA IP, the TLP header information is not accessible from

user logic. Therefore, the configuration space registers and the logic to build correct response TLPs are implemented within the IP itself by modifying the IP source files.

Forming the Capability List

The next challenge is correctly forming the PCI capability list with the new Virtio capabilities. The capability list is traversed by following the "next" pointer of each capability entry. Since the PCIe integrated block implements power management, PCIe, Message Signaled Interrupts (MSI), and MSIx capabilities, the next pointer of the last of those capabilities should be set to point to the first Virtio capability. Surprisingly, the Vivado IP flow does not expose this option to the user even though configuration space access forwarding is exposed. We noticed that there are potentially multiple ways to set this pointer, but later discovered that not all methods satisfy our requirements.

Using the DRP Interface: Depending on the capabilities chosen when generating the IP core, one of the attributes, `MSIX_CAP_NEXTPTR`, or `PCIE_CAP_NEXTPTR` should be set to point to the first Virtio capability using the DRP interface. However, this method is limited by the fact that user logic may not be operational in time to make this change visible at the time of device enumeration.

RTL modification: It is also possible to modify the IP source code directly to set the above attributes as parameters when instantiating the PCIe integrated block. Since we expect the FPGA to be correctly enumerated at system boot up, this is our preferred option to set the capability list next pointers.

4.2.3 Virtio structures

The Virtio structures pointed to by the newly added PCI capabilities are used in both device initialization and regular operation. We briefly describe our implementation here. As we have successfully added Virtio capabilities to the PCI capability list, we are free to place the Virtio data structures on any of the BARs. since the XDMA IP core provides an AXI lite master interface with the user logic that is mapped to BAR0, we place all the Virtio structures on BAR0 at different offsets. Each of the Virtio capabilities informs the driver of both the BAR and the offset the corresponding data structure is located at. The Virtio driver will perform read and write accesses to BAR0 to access the common configuration, notification, ISR status, and device-specific configuration structures. The PCI configuration access capability does not point to a different structure on a memory or I/O region. Instead, the driver can access other Virtio structures by reading/writing configuration space addresses corresponding to this capability.

Common Configuration Structure: Common configuration structure perhaps is the most important of the Virtio structures. First, it informs the driver of device attributes such as the features offered, the number of queues supported, and the size of each queue. The driver writes information such as the addresses for different virtqueue regions, and the MSIx vectors for each queue to the common configuration structure. The device and driver feature fields are used in feature negotiation between the device and the driver. The `'device_status'` field has bits indicating different stages of device initialization. Some fields in the common configuration structure represent a group of replicated fields. For instance, all fields referring to individual queues are replicated to match the number of queues supported. The `'queue_select'` field determines which queue is being configured. The common configuration structure

implementation should include the necessary logic to support this behavior.

Notification Structure: The driver writes to the notification structure to notify the device when there are new buffers added to the queues. Because the driver never reads the notification structure, it is not implemented using registers. Instead only the logic necessary to respond to write requests is implemented. The controller FSMs for virtqueues monitor the writes to addresses corresponding to the notification structure and initiate data movement between the host and the device as necessary. Depending on the features negotiated, the driver may either write to the same address or write to a different address within the notification structure that corresponds to each queue. The data written includes which queue the notification is for. Therefore, the virtqueue control logic can differentiate notifications for each queue.

ISR Status Structure: ISR status field is used by the driver to differentiate between queue and device configuration interrupts. This field is only useful when using legacy `INTx` interrupts. This field is implemented as a register that gets cleared on read.

Device-Specific Configuration Structure: 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 calculation task for the incoming packets is offloaded to the device.

PCI Configuration Access Capability: The driver is provided with an alternative access mechanism to the common configuration notification, ISR status, and device-specific configuration structures through this capability. This functionality is achieved through further modifications to the logic that implements the modified PCI capability list. A new state machine is added to capture configuration space read-/write TLPs intended for PCI Configuration Access Capability, and issue a read or write request as necessary to the module that implements the Virtio structures. The portion of the PCI configuration space that corresponds to the PCI Configuration Access Capability is generally readable and writable by the host. Reads/writes to the `'pci_cfg_data'` field of the capability trigger this special behavior. The values of the rest of the fields determine which of the structures is accessed via this alternative mechanism.

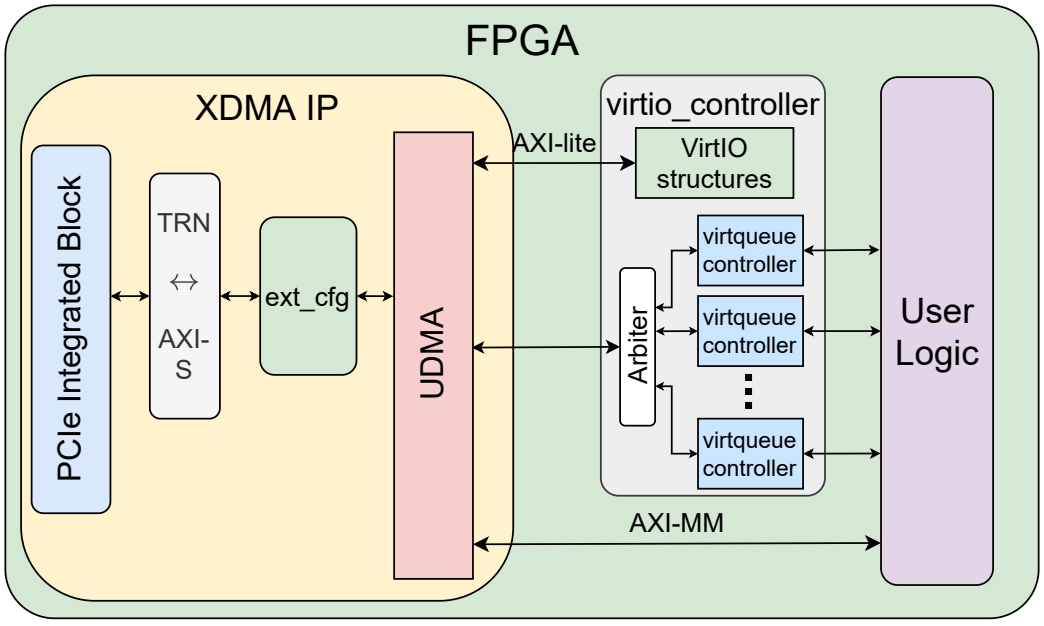


Figure 4·1: Virtio implementation.

The connectivity between different modules is shown in Figure 4·1. Here, the UDMA module is a sub-IP that implements the DMA functionality. The source code

for this module is not visible to the user. All the AXI interfaces going to user logic originate from this module. Please note that the block named $TRN \leftrightarrow AXI{-}S$ is not an actual sub-module of the XDMA IP. Rather it represents the functionality of multiple modules that sit between the PCIe integrated block and the UDMA module and perform conversions between the 'transaction interface' of the integrated block and the AXI stream interface connected to the UDMA module. Similarly, the block named 'ext_cfg' represents modifications made to multiple modules of the IP core to implement a part of the PCI configuration space external to the PCIe integrated block. We may refer to 'ext_cfg' as a single module from now on for simplicity.

When a configuration space access is forwarded from the PCIe integrated block, the 'ext_cfg' logic intercepts the TLP and responds to the request. The modified PCI configuration space registers with Virtio capabilities are part of 'ext_cfg.' Memory read and write TLPs are not intercepted and are sent directly to the UDMA module. When a configuration space read/write TLP intended for the PCI Configuration Access Capability is received, the 'ext_cfg' logic still intercepts the TLP. Instead of immediately responding, it converts the TLP to a memory read/write TLP and sends it to the UDMA module as any other memory access TLP. The fields of PCI Configuration Access Capability are used to modify the fields of the TLP. It also indicates to the UDMA module that BAR0 was accessed. This results in the UDMA module sending a read/write request to the 'virtio_controller' module which is connected to the AXI-lite interface and includes the Virtio data structures. The response from the 'virtio_controller' is sent to the host.

The sequence of transactions involved in PCIe device enumeration and Virtio device initialization is shown in Figure 4·2. In the device enumeration phase, the host OS kernel reads the configuration space of the PCIe device to identify the device. The accesses below a predefined address (`'hA8` for our implementation targeting the Artix

7 device) are handled by the PCIe integrated block. As part of the configuration space is implemented externally to the PCIe integrated block, the 'ext_cfg' logic services the accesses above the configuration space address `'hA8`. After allocating memory regions corresponding to the BARs of the device, the addresses of the memory regions are written to the BARs. Virtio device initialization begins when the kernel PCIe code calls the *probe* function of the Virtio driver. The device driver first reads the Virtio structures to gather information about the device such as the number of queues supported, and the size of each queue. Next, the driver writes the addresses for the different queue data structures in host memory to the common configuration structure on the device. It also updates the device status register at each step of the initialization.

4.2.4 Data Movement

Data movement between the host and a Virtio device is done using virtqueues. While the Virtio specification describes two different virtqueue formats, we have implemented the simpler *split virtqueue* definition. It should be noted that there are no limitations from the FPGA for a *packed virtqueue* implementation. We have made our selection purely based on the simplicity of the split virtqueue functionality.

A split virtqueue consists of three regions. These are the available ring, used ring, and the descriptor table. All of these are data structures in the host memory. The starting addresses for each of these regions, for each of the queues are written to the common configuration structure at device initialization. Therefore, when a notification is received for a given queue, the virtqueue controller modules can start data movement without further interventions from the host. We have opted to implement individual controllers for every queue supported by the device. The interface to control the DMA engine is shared between all the virtqueue controllers as shown in Figure 4·1.

When data is available to be moved from the host to the device, the Virtio driver places data in a buffer, updates entries in the available ring and descriptor table of the TX virtqueue, and notifies the device by writing to the notification structure. The virtqueue controller takes over the data movement at this point. It first accesses the header of the available ring to figure out how many ring entries were created by the driver. The valid entries are traversed one by one while doing the following;

1. Read available ring entry to determine the corresponding descriptor table entry.
2. Read a descriptor from the descriptor table.
3. Use the descriptor to access the buffer and move data to the device.
4. Create an entry in the used ring to indicate that an entry from the available ring was completed.
5. Update the header fields of the used ring to resemble the new number of entries in the used ring.
6. Notify the driver via an interrupt.

Each of the first five steps corresponds to the virtqueue controller programming the DMA engine to perform a data movement. The interrupt could be either a legacy interrupt or an MSIx interrupt. We have opted to use MSIx interrupts in our implementation. If sufficient MSIx vectors are enabled, the driver may choose to assign different vectors for each queue and one vector for configuration changes. The virtqueue controllers are responsible for selecting the correct MSIx vector. The vector number corresponding to each queue is written to the `'queue_msix_vector'` field of the common configuration structure at device initialization.

The Virtio driver allocates buffers for the RX queue as well and notifies the device immediately after the device initialization. However, these are not used until the user

logic asserts an input signal to the virtqueue controller corresponding to an RX queue. The virtqueue controller exposes an interface to the user logic to indicate the source address to read data from and the length of the buffer. When the user logic indicates that data is ready to be moved to the host, the virtqueue controller follows the same steps as when it received a notification for the TX queue. The only difference is that instead of moving data from host to device, it moves the data from device to host into a buffer previously allocated by the Virtio driver.

Controlling the DMA engine

The use model for most DMA engines involves a device driver running on the host providing the device with the address where the descriptors are stored, the DMA engine fetching the descriptors, and then performing the data movement. The descriptor layout is specific to the DMA engine. Since the Virtio drivers do not target a specific device, the descriptors in the descriptor table do not match the layout required by the DMA engine of the XDMA IP core. Furthermore, the Virtio driver does not attempt to program the DMA engine at all. Instead, it provides the device with the starting addresses of different regions of the virtqueues. The device has to use these and the information on the size of the ring and descriptor table entries available in the Virtio specification to control the DMA engine and perform data movement.

The XDMA IP core provides an interface named the "Descriptor Bypass Interface" to feed descriptors to the DMA engine from user logic. The virtqueue controllers use this interface to program the DMA engine and move data to and from the FPGA. The "DMA status ports" are used to monitor the completion of data movement and advance the states of the virtqueue controller FSM. While we expect most PCI/DMA IP cores for different FPGAs to have similar interfaces to control the DMA engine from user logic, the user might have to implement their own DMA engine if such an

interface is not available.

Figure 4·3 depicts the transactions involved in the regular operation of a Virtio device. The red-colored arrows represent the Virtio controller programming the DMA engine. The directions of the DMA transfers are indicated within parentheses as `C2H` and `H2C` for card-to-host and host-to-card respectively. Finally, the diagram is drawn assuming that the interface between the Virtio controller and the user logic is the AXI memory-mapped variant of the interface. If the Virtqueue variant is enabled, more transactions are required to update the virtqueue rings and descriptor table in the FPGA memory.

4.3 Evaluation

4.3.1 Results

Our implementation of the FPGA-based Virtio console device is implemented using an Alinx AX7A200 development board. It has been tested on a host machine running Fedora 37 operating system. We have shown that with the added hardware support for Virtio, the device gets enumerated as a Virtio console device at system boot up without requiring additional PCI bus rescans. Also, the Virtio driver loads completely without any errors and is agnostic to the fact that it is communicating with a physical device and not a virtual device. Figures 4·4, and 4·5 present the output of `lspci -v` on the host machine with the FPGA programmed with bitstreams for a Xilinx example PCIe DMA design and the Virtio console device respectively. The PCI capability list in Figure 4·5 shows the Virtio capabilities added by modifying the PCIe IP core.

We evaluate the performance of Virtio drivers further in Chapter 5. Our testing shows that the data movement performance is similar to or marginally better than the vendor-provided reference driver. Furthermore, we have observed a lower variance in

latency when using Virtio drivers. These observations can be explained by the steps involved in data movement and the way the DMA engine is programmed when using vendor-provided versus Virtio drivers.

Host to Card (H2C): To perform an H2C transfer, the legacy driver programs the DMA engine through multiple memory read and write accesses over PCIe. When the transfer is complete, the DMA engine interrupts the driver. In contrast, the Virtio driver only performs one memory write to the Virtio notification structure of the device. While the virtqueue controller has to perform multiple reads and writes to host memory, these are DMA transfers and the DMA engine is programmed directly from within the FPGA. The DMA engine can be programmed in a single clock cycle via the descriptor bypass interface unless the engine is already busy moving data. We expect this to have a lower latency compared to the multiple memory reads and writes necessary for the legacy driver.

Card to Host (C2H): When data is ready to be sent back to the host and the legacy driver is being used, the FPGA has to first raise an interrupt to indicate to the driver that data is ready to be moved. Then the driver will program the DMA engine similar to the H2C scenario. If the driver has to figure out the source address on the FPGA by reading a CSR or a similar mechanism, that adds more memory reads to the total. When the data movement is complete, the DMA engine interrupts the driver.

As opposed to this, the operation of the virtqueue is much more streamlined. When user logic indicates to the virtqueue controller that data is ready to be sent to the host, the controller determines the buffer location in host memory by reading the available ring and descriptor table of the virtqueue. These are DMA operations and have lower latency compared to the memory accesses by the legacy driver. Then the

data is moved to the buffer in host memory, and finally, the Virtio driver is notified via an interrupt. We expect this to have a lower latency than the legacy driver operation because of faster DMA operations and one less interrupt. If legacy INTx interrupts are used, the Virtio driver has to perform one more read operation to the ISR status field. However, it is not necessary if MSIx interrupts are used because each virtqueue can be assigned a unique interrupt vector.

4.3.2 Resource Usage

The `virtio_controller` module is central to implementing a Virtio-compliant interface on the FPGA. However, it uses additional FPGA resources compared to a design that uses only the vendor-provided IP core and the device driver. Tables 4.1, and 4.2 show how the resource usage for the `virtio_controller` module increases with the number of queues supported. Resource usage is represented by the total LUT and Flipflop (FF) usage. Because the `virtio_controller` module does not use BRAMs, we do not consider the BRAM usage of the XDMA IP core for this analysis. Neither module uses DSP slices.

Queues	XDMA	virtio_controller
2	14,250 (10.65%)	6,907 (5.16%)
4	Same	8,643 (6.46%)
8	Same	12,118 (9.06%)
16	Same	19528(14.59%)

Table 4.1: Resource usage (total LUTs) comparison between the PCIe IP core and the Virtio controller

Queues	XDMA	virtio_controller
2	15,721 (5.87%)	4,909 (1.83%)
4	Same	5,799 (2.17%)
8	Same	7,873 (2.94%)
16	Same	11,866 (4.43%)

Table 4.2: Resource usage (total Flipflops) comparison between the PCIe IP core and the Virtio controller

Around 8 queues, the `virtio_controller` module's LUT usage approaches the resource usage of the XDMA IP core, essentially doubling the resources used to implement host-FPGA PCIe communication. The FF usage grows at a lower rate because the Virtio structures which use most of the flipflops are common to all the queues supported by the device.

4.3.3 Implementation challenges

In this section, we provide our assessment of the difficulty of implementing Virtio support on an FPGA by modifying vendor IP blocks. According to our estimates, over 80% of the implementation effort was put into figuring out the necessary features of the vendor IP cores and using those features to add Virtio capabilities to the PCI capability list. Implementing the Virtio data structures and virtqueue controller logic according to the Virtio specification was straightforward and quick. If the PCIe IP core in question is one used with more than one device family, the effort to figure out the IP details is not necessary when implementing Virtio support on the other device families. For instance, the Xilinx XDMA IP core used in this work can work with UltraScale+, UltraScale, Virtex-7 XT Gen3 (Endpoint), and 7-series Gen 2 (Endpoint) Integrated Blocks for PCIe [Xilinx, 2022b]. Therefore, implementing Virtio support on those devices can be done with considerably less effort. Also, the FPGA vendors can enable the necessary capabilities to make Virtio support easily achievable.

4.4 Integrating into DISL

When adding a component to the DISL component library, we should follow the steps described in Section 3.3.1 to improve design reuse and portability across devices. The main task to be performed is separating the device-specific and generic logic of the new component. This helps improve design reuse and portability across devices.

As shown in Figure 4·1, the main components of the PCIe subsystem are the

vendor-provided PCIe IP and the Virtio controller module. Out of these, the latter only consist of generic logic. Therefore, it belongs in the `Generic` code partition. The PCIe IP core includes both device-specific and generic logic. The 'PCIe Integrated Block' is a hardened ASIC block that implements the functionality of the PCIe interface. This includes the physical and link layers and the PCI configuration space. As this is specific to a given device, this belongs in the `Board` partition. In contrast, the block named 'UDMA' is a sub-IP that acts as a DMA engine. It is possible for a user to replace this with their own DMA engine. The interface logic blocks between the PCIe integrated block and the DMA engine are implemented on the FPGA fabric using RTL (Note that Figure 4·1 shows a simplified view of the logic between the PCIe integrated block and the UDMA block). However, some of the blocks can be categorized as device-specific since they implement interfaces that match the integrated block. A strict partitioning scheme would categorize individual modules into `Generic` and `Board` partitions.

However, such a partitioning scheme does not provide a significant benefit over one that categorizes the complete IP core as belonging to the `Board` partition due to a few reasons.

1. A partitioning scheme that separates individual modules of the IP core into `Board` and `Generic` partitions would probably require some of the IP logic including the interfaces between the modules now separated into partitions to be reworked. This requires a deep understanding of the IP core's architecture and considerable effort.

2. When porting to a new device, we have to generate a new IP core that matches the new device.

    - It is straightforward to replace the whole IP.

- Since IP cores differ from one another significantly, most modifications done to the sub-modules of the IP core need to be repeated.

3. The modifications we made to enable the features required to implement a Virtio interface are not extensive to the point that we remove/replace complete sub-modules of the IP core. We are only making minor modifications to the modules.

4. Most users don't have any restrictions regarding using vendor-provided IP cores.

Therefore, considering that a stricter partitioning scheme:

1. Does not improve portability or reuse more than a simpler partitioning scheme

2. Requires significantly higher effort

it generally does not make sense to use one. However, there could be situations where even more extreme partitioning makes sense. For instance, consider a user who does not wish to use any vendor-provided IP cores or a user who wishes to obtain higher performance than what the vendor-provided IP core can provide. Even in such a use case, the PCIe integrated block still has to be used because it is a hardened ASIC block without which PCIe connectivity cannot be implemented. Every other piece of logic in the IP core can be replaced by a designer with generic logic. There should be a compelling reason such as the ones mentioned above to go for an extreme partitioning scheme.

Figure 4·6 shows different partitioning schemes we could implement with the PCIe subsystem. The new functionality added (Virtio) is mostly implemented as generic logic and any changes to the IP core are contained within the board-specific RTL of the IP core. Therefore, partitioning the IP core's RTL into separate code partitions does not improve the design portability. Hence, we have followed the partitioning

scheme (3). The ‘pcie_controller’ module is added to the Generic code partition while the modified PCIe IP core is added to the Board partition. Code in the Board partition includes modified RTL source files for the IP core. When the IP core is generated, some of the IP source files are replaced with the modified ones.

One minor drawback of partitioning scheme (3) is that the Board and Generic partitions are connected over the interfaces presented by the IP core. Since different IP cores can use different interfaces, it may be necessary to implement additional glue logic to ensure compatibility between the reused Generic components and new Board components when porting to a new device. The other partitioning schemes provide more control over the interface between Board and Generic partitions. However, the effort necessary to implement glue logic is most likely less than the effort to modify the IP core in the other two schemes.

A potentially bigger issue is the interfaces of an IP core lacking the functionality to support the Virtio functionality. In this case, the only viable options are to modify the IP core or use a different IP core. If modifying the IP core, partitioning schemes (1) or (2) could be used depending on the extent of the modifications.

4.4.1 Updating Configuration Files

DISL uses configuration files written in TOML syntax to describe components and systems. After partitioning our design into device-specific and generic portions, we have to add these to the corresponding configuration files before using the PCIe subsystem in DISL. Our initial implementation was using an Alinx ‘AX7A200T’ development board. This uses a Xilinx Artix 7 (XC7A200TFBG484-2) FPGA. The parts of the design that belong to the Board partition are specified in /fpga/boards/alinx_ax7a200t/config/board.tml file.

```
1  ...
2  [REQUIREMENTS.FILES."xdma_xc7.sv"]
3    HDL = ["xdma_0_axi_stream_intf.sv", "xdma_0_pcie2_ip_core_top.v", "
       xdma_0_rx_demux.sv", "xdma_0_tgt_cpl.sv", "xdma_0_tgt_req.sv"]
```

```
  IP = []
...
[REQUIREMENTS.IP]
[REQUIREMENTS.IP."xdma_xc7.sv"]
xdma_0 = """
    config_ip_cache -disable_cache
    create_ip -name xdma -vendor xilinx.com -library ip -version 4.1 -
    module_name xdma_0
    set_property -dict [list ...
    <A long list of configuration parameter values>
    ...] [get_ips xdma_0]
    generate_target all [get_files ./build/[lindex $argv 0]/[lindex $argv 0].
    srcs/sources_1/ip/xdma_0/xdma_0.xci]
    export_ip_user_files -of_objects [get_files ./build/[lindex $argv 0]/[lindex
     $argv 0].srcs/sources_1/ip/xdma_0/xdma_0.xci] -no_script -sync -force -
    quiet
    create_ip_run [get_files -of_objects [get_fileset sources_1] ./$PROJECT/
    $PROJECT.srcs/sources_1/ip/xdma_0/xdma_0.xci]
    launch_runs xdma_0_synth_1 -jobs 24
    wait_on_run xdma_0_synth_1
    update_compile_order -fileset sources_1
    set_property IS_LOCKED true [get_files xdma_0.xci]
    exec cp ./xdma_0_axi_stream_intf.sv ./$PROJECT/$PROJECT.srcs/sources_1/ip/
    xdma_0/xdma_v4_1/hdl/verilog/xdma_0_axi_stream_intf.sv
    exec cp ./xdma_0_pcie2_ip_core_top.v ./$PROJECT/$PROJECT.srcs/sources_1/ip/
    xdma_0/ip_0/source/xdma_0_pcie2_ip_core_top.v
    <Copy more files>
    reset_run xdma_0_synth_1
    launch_run xdma_0_synth_1
    wait_on_run xdma_0_synth_1
    """
...
[IO]
  [IO.sys_clk_p]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "CLOCK"
  [IO.sys_rst_n]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "GENERAL"
  [IO.pci_exp_txp]
    DIRECTION = "SINK"
    WIDTH = 4
    INTERFACE_TYPE = "GENERAL"
  <More interface definitions>
...
[CONSTRAINTS]
sys_clk_p = """
set_property LOC IBUFDS_GTE2_X0Y3 [get_cells refclk_ibuf]
create_clock -period 10.000 -name sys_clk [get_ports {sys_clk_p}]
"""
sys_rst_n = """
```

```
50  set_property -dict {PACKAGE_PIN L16   IOSTANDARD LVCMOS18  PULLUP true} [
        get_ports {sys_rst_n}]
51  set_false_path -from [get_ports sys_rst_n]
52  """
53  ...
```

Listing 4.1: Adding PCIe components to board.tml

Listing 4.1 shows the parts of the `board.tml` that are related to the main PCIe design components. Line 4 specifies five HDL files as dependencies for the `xdma_xc7.sv` file. These are RTL sources of the PCIe IP core modified to add new functionality that helps us implement the Virtio interface. These are placed in the `/fpga/boards/alinx_ax7a200t/src/hdl` directory. When a design that uses the PCIe subsystem is generated, these are copied to the build directory. Lines 10-28 include commands to generate the PCIe IP core and modify it. Note that the IP is compiled once and then some of the source files are replaced and recompiled. This is a tool-specific sequence of operations necessary because of how Xilinx Vivado generates and manages IP source files. These commands are added to the tool-specific scripts when the design is generated. Therefore, a user does not have to interact with these tool-specific details when using the PCIe subsystem.

Starting at line 32 in Listing 4.1, several new board I/O signals are added to the `[IO]` dictionary. Starting on line 41 (and not shown completely) are the PCIe TX and RX lanes. Note that these are four lanes. The `sys_clk_p` and `sys_rst_n` are clock and reset signals from the PCIe slot. The constraints for the clock and reset signals are added to the `[CONSTRAINTS]` dictionary.

```
1  ...
2  [xdma_xc7]
3    TYPES = ["PERIPHERAL"]
4
5    PARAMETERS = []
6
7    [xdma_xc7.REQUIREMENTS]
8      INTERFACES = ["sys_rst_n", "sys_clk", "pci_exp_txn", "pci_exp_txp", "
       pci_exp_rxn", "pci_exp_rxp", "m_axi", "m_axi_bid", "m_axi_rid", "m_axil", "
       s_axil", "s_axil_awprot", "s_axil_arprot", "c2h_dsc_byp", "h2c_dsc_byp", "
       c2h_sts_0", "h2c_sts_0", "usr_irq_req", "usr_irq_ack", "axi_aclk", "
```

```
      axi_aresetn"]
      [xdma_xc7.REQUIREMENTS.INCLUDES]
        COMMON = []
        BOARD  = ["xdma_xc7.sv"]
    [xdma_xc7.ENCODINGS]
    [xdma_xc7.INTERFACES.sys_rst_n]
      TYPE = "GENERAL"
      WIDTH = 1
      DIRECTION = "SINK"
    [xdma_xc7.INTERFACES.sys_clk]
      TYPE = "CLOCK"
      WIDTH = 1
      DIRECTION = "SINK"
    [xdma_xc7.INTERFACES.pci_exp_txn]
      TYPE = "GENERAL"
      WIDTH = 4
      DIRECTION = "SOURCE"
    [xdma_xc7.INTERFACES.pci_exp_txp]
      TYPE = "GENERAL"
      WIDTH = 4
      DIRECTION = "SOURCE"
    [xdma_xc7.INTERFACES.pci_exp_rxn]
      TYPE = "GENERAL"
      WIDTH = 4
      DIRECTION = "SINK"
    <A long list of interfaces>
  ...
```

Listing 4.2: Adding PCIe IP's top module to board.tml

Next, a description of the top module of the IP core is added to `'/fpga/common/config/modules.tml'`. A part of the module description in the `modules/.tml` file is shown in Listing 4.2. All the interfaces of the module are described here. Note that `'xdma_xc7'` is a wrapper module that encapsulates the actual top module of the PCIe IP generated by the FPGA toolchain. This wrapper is implemented to differentiate between different versions of the PCIe IP core. The XDMA IP core has different variants designed for different devices. The top module for the IP generated by the toolchain is always named `'xdma_0'` regardless of which version of the IP it is. The wrapper modules help distinguish between the different versions of the IP core.

After adding the device-specific portion of the PCIe subsystem to DISL, we are

left with the `Generic` partition which includes the `pcie_controller` module. The next step is to add that to the `modules.tml` file. It is added to the configuration file similarly to the `'xdma_xc7'` module in Listing 4.2.

4.5 Conclusion

In this chapter, we presented the requirements for enabling host-FPGA communication using native Virtio drivers. We have identified and implemented the data structures, arbitration logic, queues, and DMA control logic required to create a Virtio-compliant host interface on an FPGA. We have done so by using both generic RTL modules and modifying the vendor-provided PCIe IP blocks. Finally, we have demonstrated the effectiveness of our approach by implementing a Virtio console device on a Xilinx 7-series device and showing correct device enumeration and communication. We believe that this work will act as a proof-of-concept for using unmodified Virtio drivers to communicate with FPGAs. It also demonstrates that even lower-end FPGAs have the necessary capabilities to implement a host interface fully compliant with the Virtio specification.

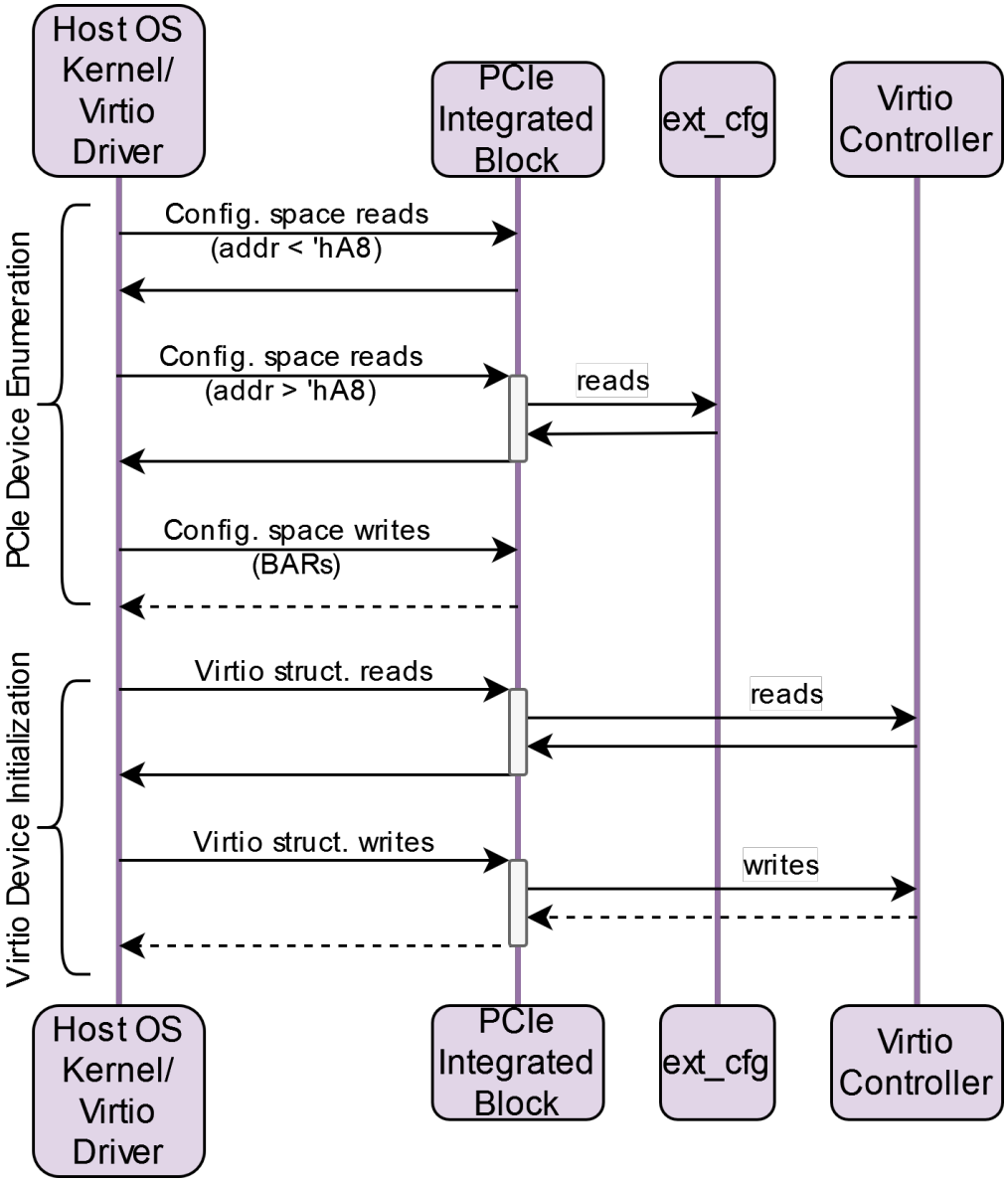


Figure 4·2: Device enumeration and initialization sequence.

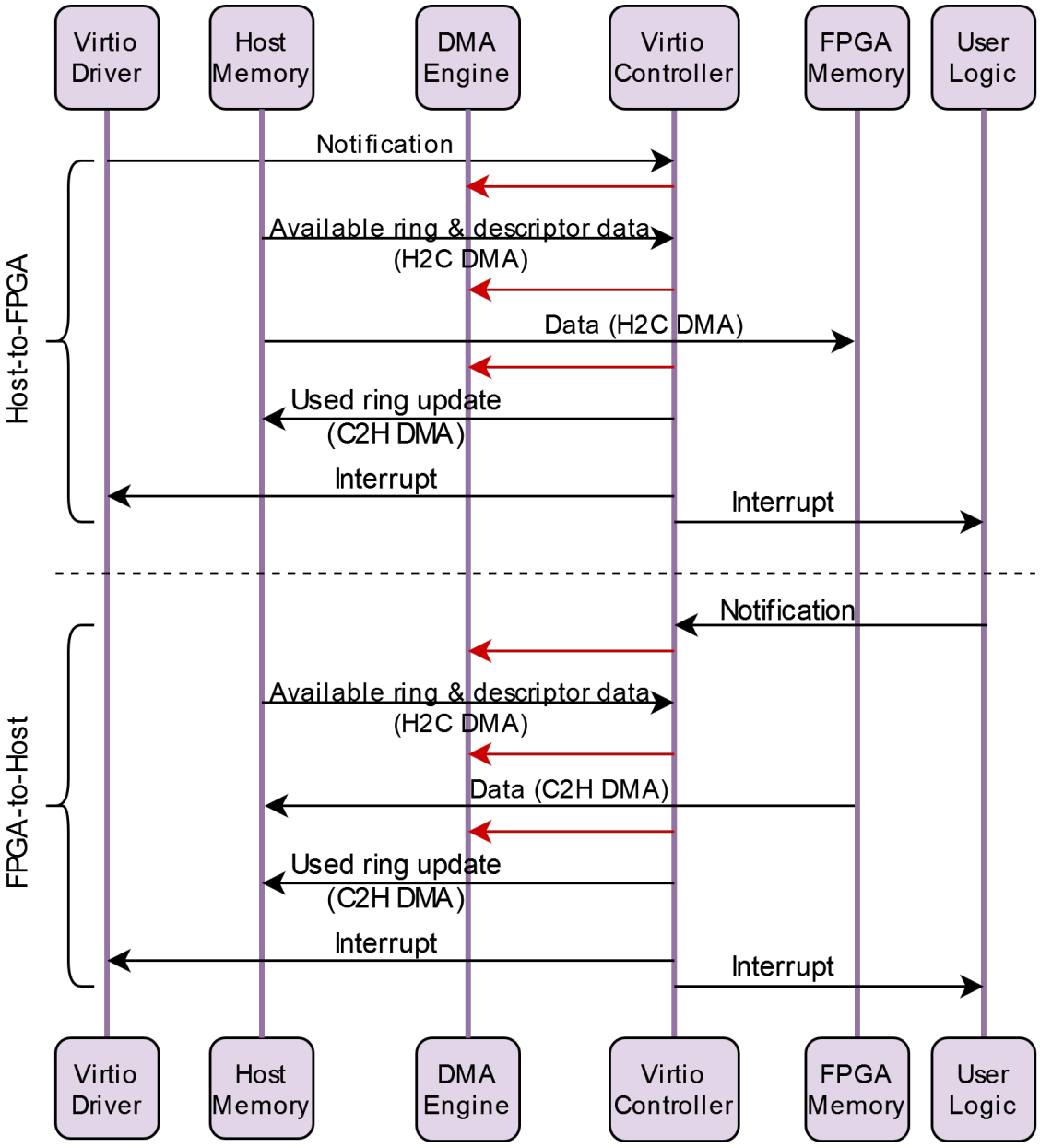


Figure 4·3: Virtio device regular operation

Figure 4·4: Device enumeration for Xilinx example design.

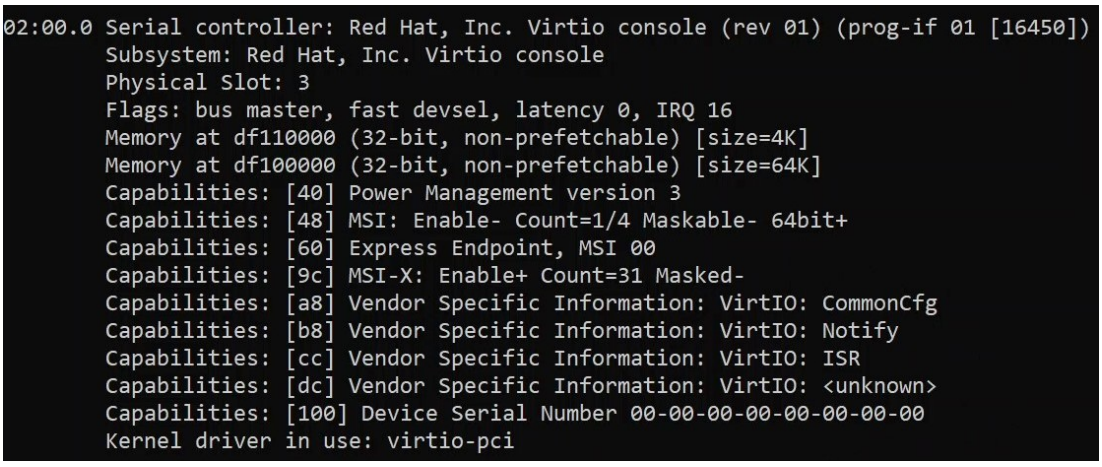


Figure 4·5: Device enumeration for Virtio console device.

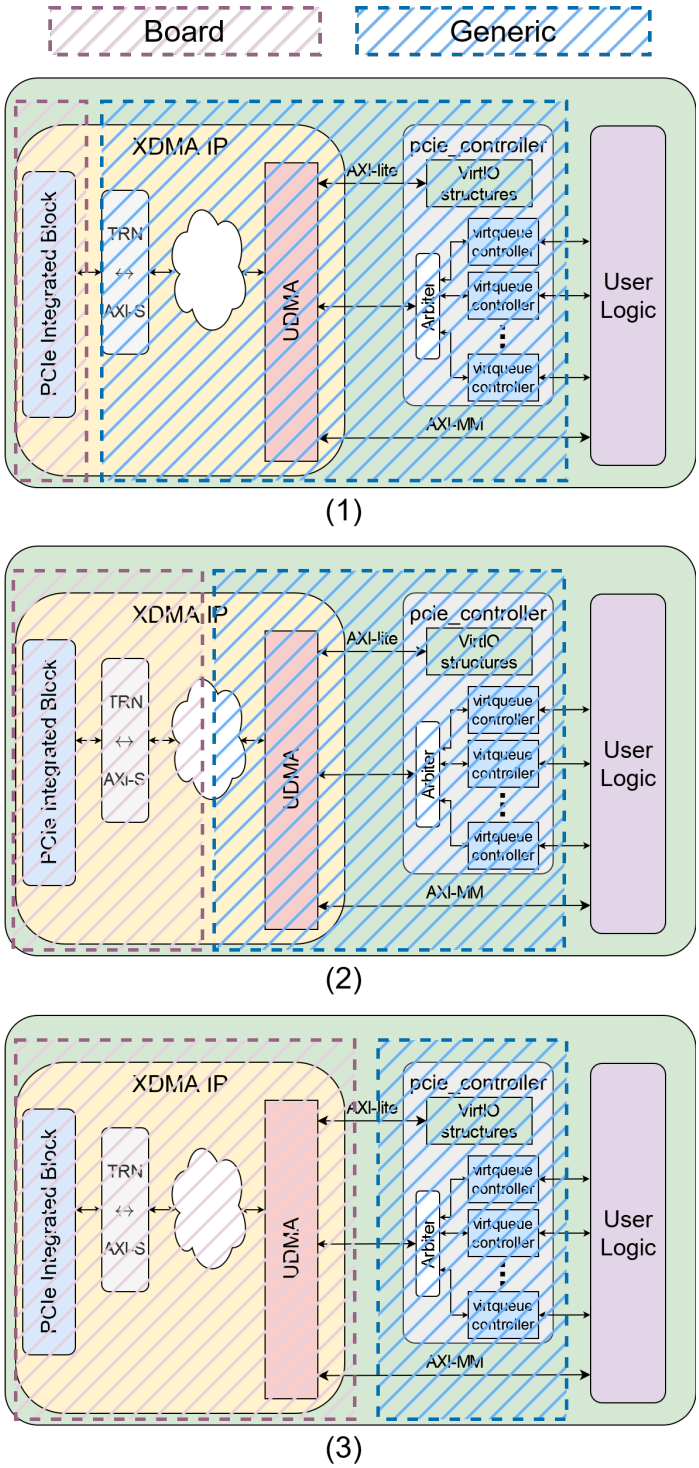


Figure 4·6: Paritioning the PCIe subsystem.

Chapter 5

Performance Evaluation and Porting the PCIe Subsystem Design

5.1 Introduction

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 the previous chapter, we proposed a general framework 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 chapter, we compare the performance of Virtio device drivers to vendor-provided device drivers in terms of communication latency and latency distribution and show that Virtio drivers provide similar or slightly improved performance with

reduced variance. To facilitate our analysis, we also extend the implementation described in the previous section by implementing support for Virtio network devices on FPGAs. A general framework to implement a Virtio interface on FPGAs should be applicable to different devices. To demonstrate the portability of our approach, in this chapter, we port the implementation from Chapter 4 to a different device. 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.

FPGAs are used as complexity offload devices for CPUs. Their inherent flexibility, combined with tight coupling of communication and computation, allows 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 [Caulfield et al., 2016, Xiong et al., 2019, Lant et al., 2020, Krishnan et al., 2021, Bobda et al., 2022, Haghi et al., 2022, Guo et al., 2022a, Wu et al., 2023]. 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 [PCI SIG Org., 2010] 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 [The Linux Kernel Archives, 2024]. Red Hat Enterprise Linux OS follows a 6-month release cycle [Red Hat Customer Portal, 2024]. 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 [Xilinx, 2024], 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 [Tsirkin and Huck, 2022] 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 section, 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.

In Chapter 4 and our published work [Bandara et al., 2022], we 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. In [Bandara et al., 2024c] we remedy that with the following contributions:

- Adding support for more Virtio device types.
- Comparing the performance of Virtio and vendor-provided device drivers using round-trip average and tail latencies.
- Highlighting the differences in device driver design, device/application semantics, and work allocation between software and hardware that impact the driver

performance and our analysis.

- Demonstrating that replacing legacy drivers with Virtio in no case results in reduced performance, but rather can even be beneficial, and often reduces variance.

5.2 Background

5.2.1 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. This allows normal bus mechanisms to be used for device discovery, interrupts, and DMA. Therefore, Virtio device drivers for use with Virtio devices treat these devices as physical devices. 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" [Tsirkin and Huck, 2022]. Virtio devices therefore: (i) use normal bus mechanisms for tasks including interrupts, DMA, and device discovery, 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 userspace 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 a physical device is 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 5·1 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.

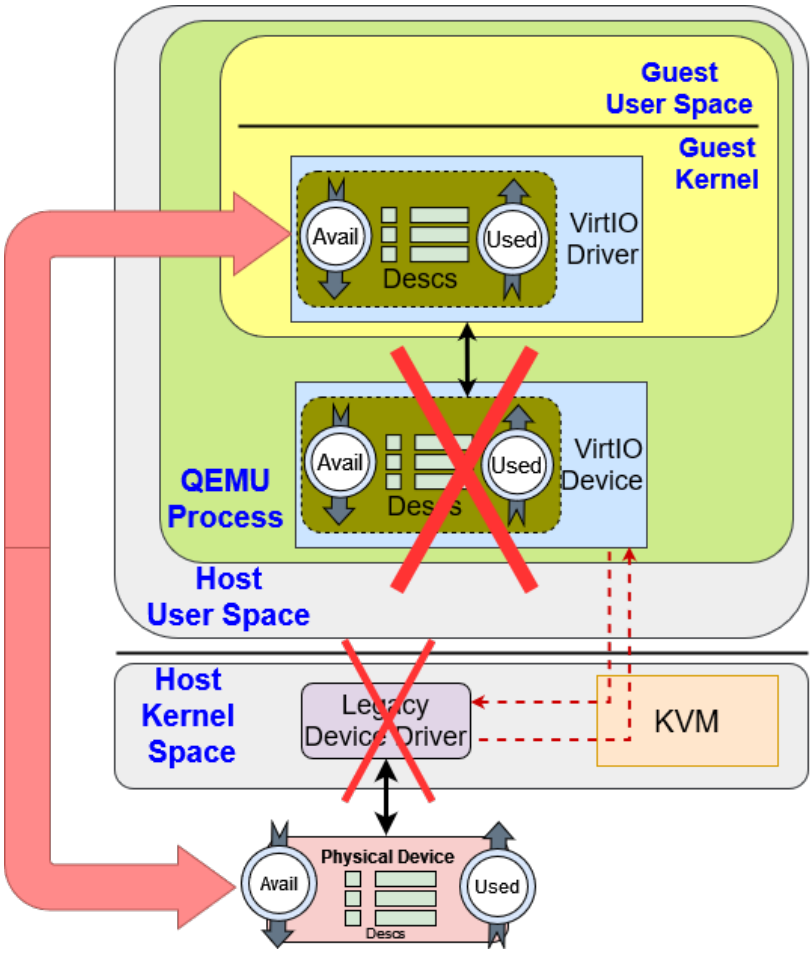


Figure 5·1: Virtio interface on the FPGA eliminates the need for back-end Virtio devices and legacy device drivers.

5.2.2 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 [Xilinx, 2024] 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.

FPGA vendors also provide runtime libraries, such as Xilinx runtime library (XRT) [Xilinx Inc., 2022] and Open Programmable Acceleration Engine (OPAE) [Intel, 2017]. 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.*

5.2.3 Using Virtio Drivers to Interact with Physical Devices

Virtio device drivers are designed with virtual devices in mind. However, 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 [Bandara et al., 2022].

5.3 Methods

5.3.1 Test case used: Virtio Network device

In this section, we extend the implementation described in [Bandara et al., 2022] 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 5·2). 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 [Tsirkin and Huck, 2022] 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.

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 are offloaded to the device. The main

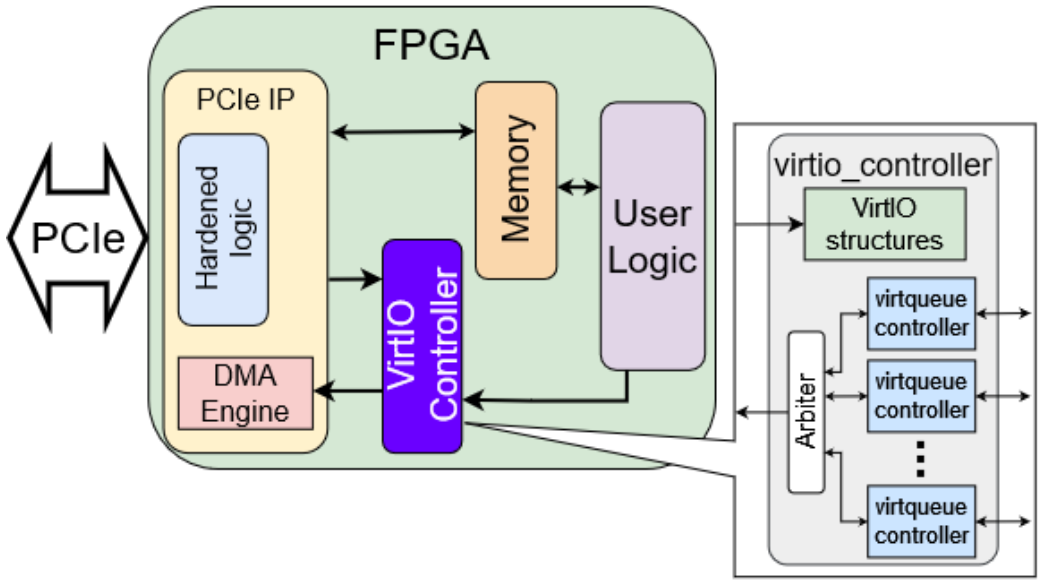


Figure 5·2: Virtio device architecture

modification to the design presented in [Bandara et al., 2022] (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 [Tahir et al., 2024a, Tahir et al., 2024b] 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.

5.3.2 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) [Xilinx, 2022b]. The host machine is running the Fedora 37 operating system.

Testing Virtio Drivers: We use the previously described Virtio network device implementation 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.

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 [Xilinx, 2024]. 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.

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.

5.4 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.

5.4.1 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 5·2).

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.

5.4.2 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 [Tsirkin and Huck, 2022].

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.

5.4.3 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.

5.5 Evaluation

This section presents and analyzes the results of the experiments described in Section 5.3.2. Figure 5·3 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 a much lower variance.

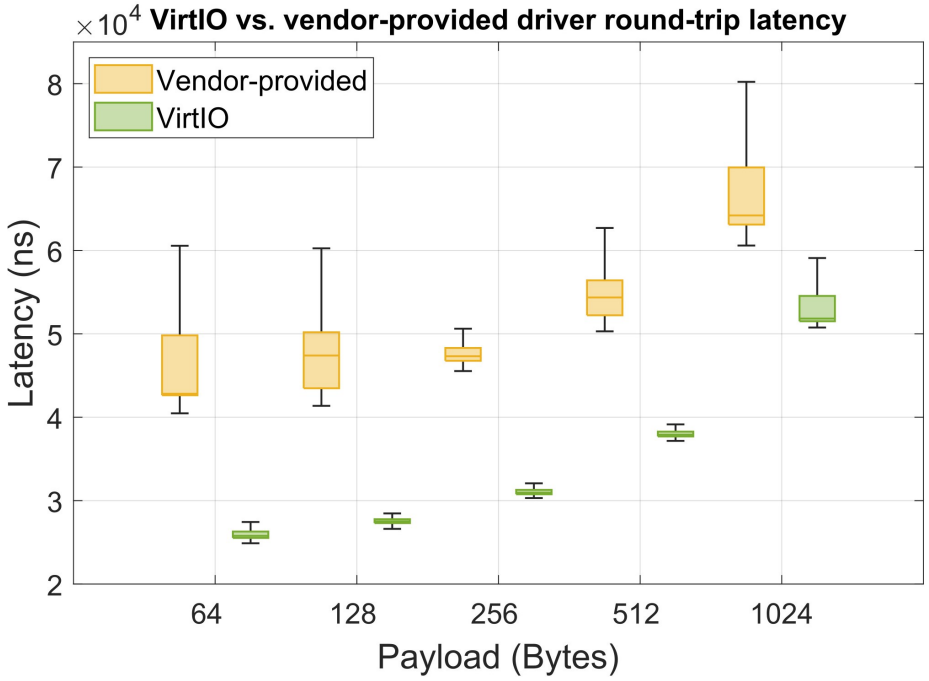


Figure 5·3: Round-trip latency with Virtio and vendor-provided device drivers.

Figure 5·4 is a visual representation of the different paths the data takes between the application and the FPGA memory when using the Virtio and vendor-provided device drivers. When using the Virtio drivers, the application uses the C socket API to send UDP packets to a set of IPs. Therefore, the operating system's network stack creates the packets using the data from the application and hands over the packets to the device driver. In the case of vendor-provided XDMA driver, the application calls the `read()` and `write()` functions of the driver to move data to and from the FPGA memory. In contrast to the Virtio case, there is no network packet creation in this scenario. Latency measurement captures the time taken to move data to the

FPGA and back to host memory. In both cases, the user application measures the latency. Therefore, the Virtio measurement also includes the overhead for creating network packets.

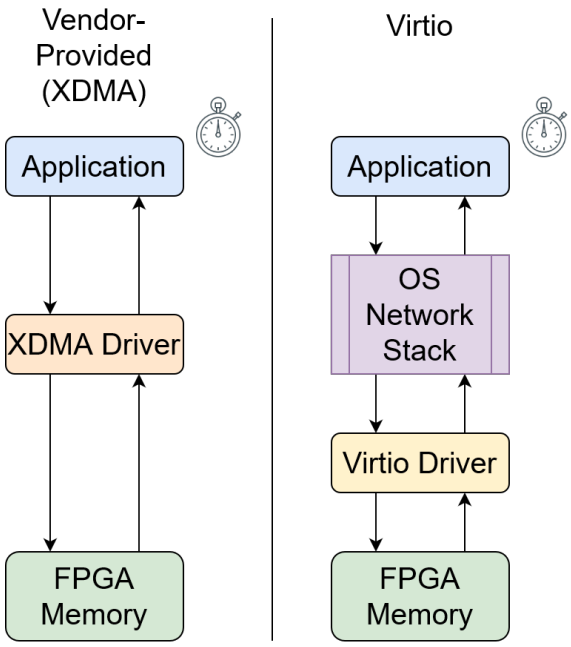


Figure 5·4: Round-trip latency visualization.

Figure 5·5 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 5·6 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.

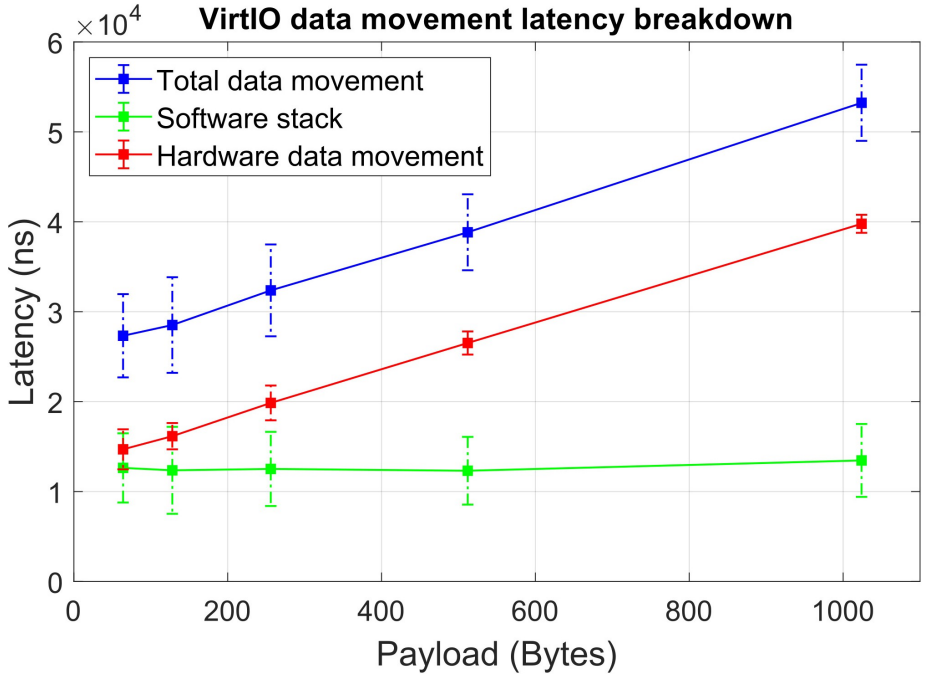


Figure 5·5: Breakdown of data movement latency using the Virtio driver.

Table 5.1 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.

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

Table 5.1: Tail latencies for data movement with Virtio and XDMA.

Our overall recommendation is as follows. For highly optimized applications with

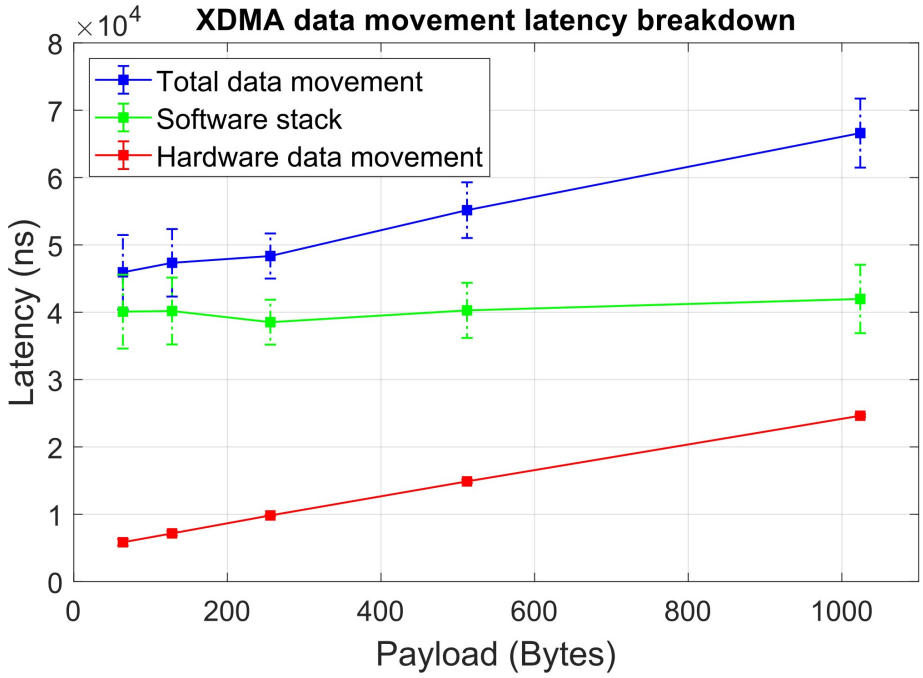


Figure 5·6: Data movement latency breakdown with the vendor-provided driver.

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).

5.6 Porting the PCIe Subsystem Design

We presented our PCIe subsystem implementation in Chapter 4 and evaluated the performance of Virtio drivers in the previous sections. It was implemented targeting an FPGA development board that uses a Xilinx Artix 7 device. When designing and implementing the PCIe subsystem, we ensured that we created a clear separation between the device-specific and generic parts of the design. In DISL parlance, the two code partitions are referred to as `Board` and `Generic`. Figure 5·7 shows how the

different modules of the PCIe subsystem are partitioned.

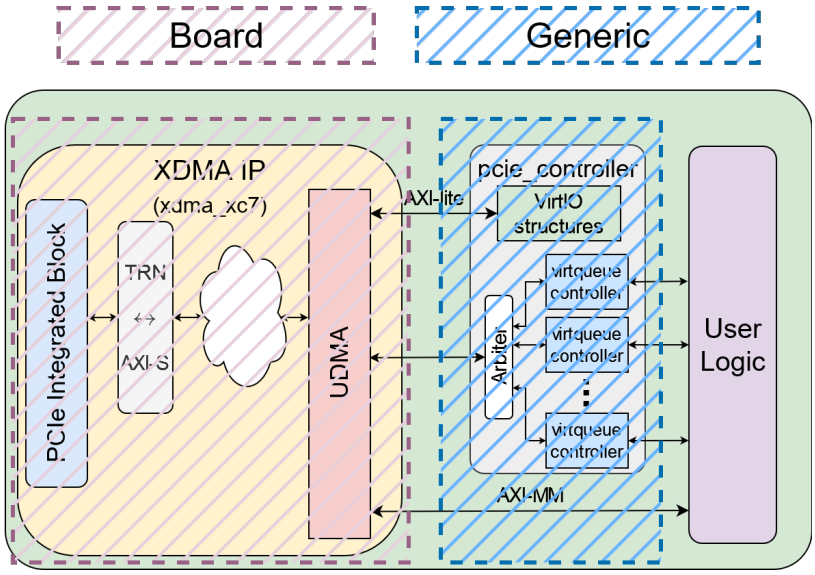


Figure 5·7: Partitioned PCIe Subsystem

In this section, we demonstrate how our design strategy makes it straightforward to port the PCIe subsystem to a different device. The new target device we have selected is a "Bittware CVP13" development board that uses a Xilinx Ultrascale+ device. To provide an understanding of the differences between the two devices, Table 5.2 compares some of the FPGA resources available on each of the devices. Apart from the FPGA devices themselves, the FPGA development boards are also significantly different as one is a low-end device while the other is a high-end device targeting high-performance applications such as cryptocurrency mining. The main difference relevant to the PCIe subsystem is that the Artix 7 development board only supports two PCIe lanes while the Ultrascale+ development board supports a full 16-lane PCIe interface.

Apart from the differences in the FPGA device and the development board, the PCIe IP cores compatible with each of the devices are also different. The IP core for the Ultrascale+ device uses the `'PCIE40E4'` integrated block that provides more fea-

Resource	xc7a200tfbg484-2	xcvu13pfigd2104-2
LUTs	133,800	1,728,000
DSPs	740	12,288
BRAM tiles	365	2,688
Gigabit Transceivers	4 (GTPE2)	76 (GTYE4)
PCIe Integrated Block	PCIE_2_1	PCIE40E4

Table 5.2: Comparison of Artix 7 and Ultrascale+ FPGAs

tures compared to the `'PCIE_2_1'` integrated block of the 7-series device. Ultrascale+ version of the IP core also provides additional capabilities such as Single Root I/O Virtualization (SR-IOV) support. The feature most relevant to our implementation is the new interface named the `''Configuration Extended Interface''`. This interface allows logic on the FPGA to respond to PCIe configuration space read and write requests from the host. This makes some of the RTL modifications we made for the Artix-7 version of the PCIe IP unnecessary for the Ultrascale+ version. Therefore, instead of those RTL modifications, we implement an external module that connects to the `''Configuration Extended Interface''` and implements a part of the PCIe configuration space external to the IP core. We still modify the IP core's RTL to correctly configure the `'next'` pointers of the PCI capability list.

Similar to the wrapper module `'xdma_xc7'` used with the previous version of the PCIe IP, we implement a different wrapper module named `'xdma_cvp13'` to instantiate the XDMA IP core and the external module named `'xdma_ext_cfg_space'`. Since this wrapper module implements the same interfaces as the `'xdma_xc7'` module, the `Generic` portion of the PCIe subsystem can be connected to `'xdma_cvp13'` module without any changes. Our design strategy ensures that the `Board` portion of the PCIe subsystem, regardless of its internal implementation details, presents consistent interfaces to the `Generic` portion. This allows the PCIe subsystem to be ported simply by replacing the `'xdma_xc7'` module with the `'xdma_cvp13'` module. Figure 5·8 depicts the differences between how the PCIe subsystem is constructed for the two FPGA devices we have used.

5.6.1 Adding Board Support

Because both versions of the PCIe subsystem use the same `'pcie_controller'` module and the interfaces between the `Board` and `Generic` partitions do not change, we do not make any changes to the `modules.tml` or `definitions.tml` files. However, the `board.tml` file is updated with new board I/O port definitions and the commands to generate the XDMA IP core for the Ultrascale+ device. Listing 5.1 shows excerpts from the `board.tml` file for the Bittware CVP13 development board to show the modifications made to support the PCIe subsystem.

```
[DESCRIPTION]
NAME = "CVP13"
DIRECTORY = "cvp13"
CHIP_VENDOR = "xilinx"
BOARD_VENDOR = "Bittware"
FAMILY.SHORT = "xcu"
FAMILY.LONG = "Ultrascale+"
PART.SHORT = "xcvu13p_0"
PART.LONG = "xcvu13p-figd2104-2-e"
...
[REQUIREMENTS.FILES."xdma_xcvu13p.sv"]
  HDL = ["xdma_0_pcie4_ip.v", "xdma_ext_cfg_space.sv"]
  IP = []

[REQUIREMENTS.IP]
[REQUIREMENTS.IP."xdma_xcvu13p.sv"]
  xdma_0 = """
    config_ip_cache -disable_cache
    create_ip -name xdma -vendor xilinx.com -library ip -version 4.1 -
    module_name xdma_0
    <more TCL commands to specify attributes, generate, modify, and regenerate>
    """
...
[IO]
  [IO.sys_clk_p]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "CLOCK"
  [IO.sys_clk_n]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "CLOCK"
  [IO.sys_rst_n]
    DIRECTION = "SOURCE"
    WIDTH = 1
    INTERFACE_TYPE = "GENERAL"
  [IO.pci_exp_txp]
```

```
37      DIRECTION = "SINK"
38      WIDTH = 16
39      INTERFACE_TYPE = "GENERAL"
40  ...
41    <PCIe TXn, RXp and RXn Lanes>
42  ...
43  [CONSTRAINTS]
44  sys_clk_p = """
45  create_clock -period 10.000 -name sys_clk [get_ports {sys_clk_p}]
46  set_property LOC AT11 [get_ports sys_clk_p]
47  """
48  sys_clk_n = "set_property LOC AT10 [get_ports sys_clk_n]"
49  pci_exp_txp = """
50  """
51  <More constraints>
```

Listing 5.1: board.tml File for the CVP13 FPGA Development Board

5.7 Conclusion

In this chapter, 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.

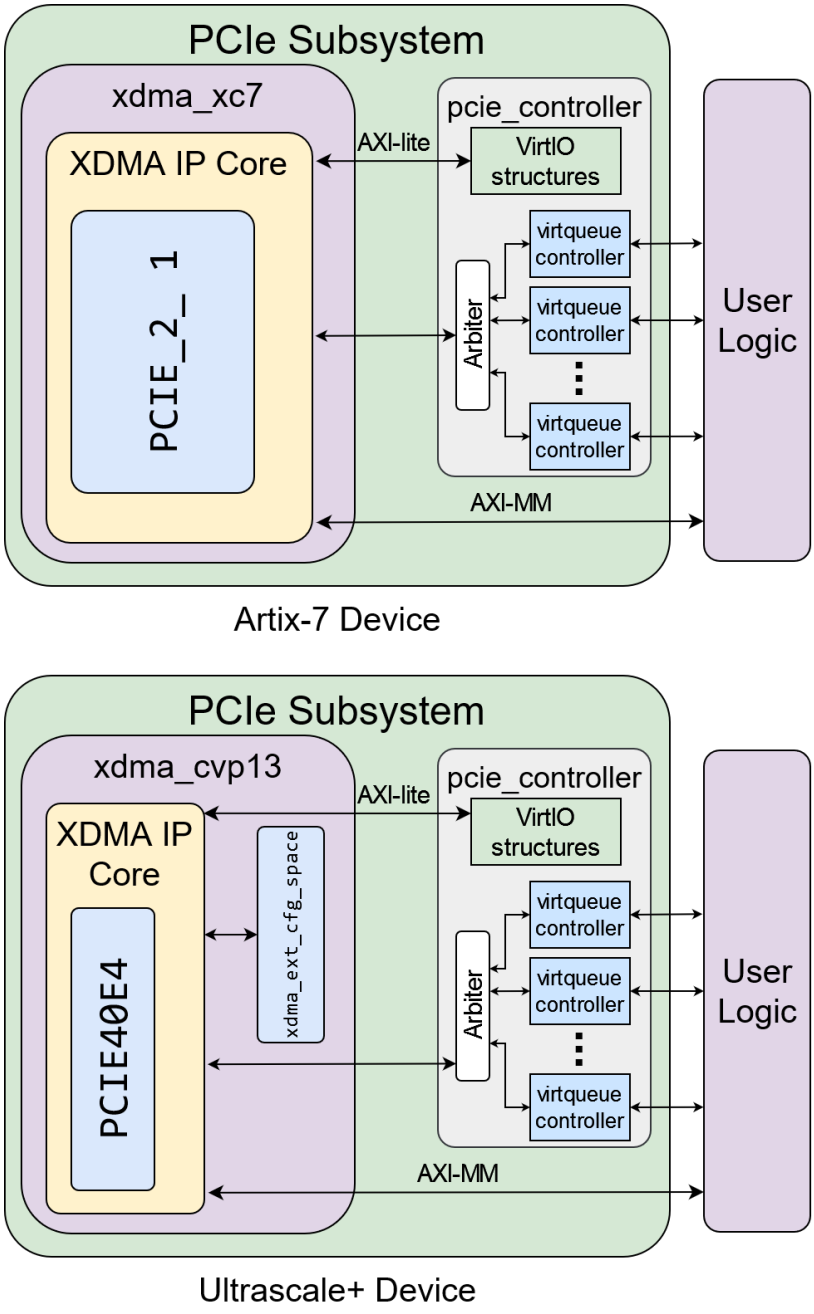


Figure 5·8: Construction of the PCIe Subsystem Using Different Variants of the XDMA IP Core

Chapter 6

Evaluating an hOS Generator

Operating system like abstractions can greatly reduce the expertise required to use FPGAs. They can also fundamentally change the FPGA use model by allowing more focused innovation where a designer can focus on perfecting the application logic rather than building the entire hardware stack. This also results in faster turnaround time and improves application portability. Using hardware operating system generators instead of fixed hOS implementations adds to these benefits. Firstly, an hOS generator reduces the effort to build a system for a particular application and port the design to different devices. A generator also allows an hOS to be customized to a specific application reducing the resource overhead. It further increases application portability to more devices.

How to evaluate a system generator such as DISL is an open question. Prior works have evaluated hOS/FPGA shells by comparing an application implemented with an hOS to a baseline design that implements the same application without using an hOS in terms of resource overhead and performance impact. However, as we are proposing an hOS generator instead of a fixed implementation, new metrics are necessary and many of those are not easily quantifiable. In this chapter, we discuss some qualitative metrics to evaluate a system generator and use those to evaluate the FPGA system generator presented in previous chapters. The metrics we discuss in this chapter are:

1. The effort necessary to build a system using the hOS generator

2. Ability to build a system that exactly matches application requirements

3. Reduced resource overhead due to customizing the system to application requirements

4. Ease of porting a design to a new device

5. Effort to modify the components and the hOS generator

We use a complete design example to aid our evaluation of the proposed hOS generator. We interleave the discussion on the hOS evaluation metrics with the design example where we implement a design using DISL and then port it to a different FPGA development board. The application is implemented as a Verilog module. All other system components used are part of the DISL component library. These include the PCIe subsystem presented in Chapter 4. We begin by describing the example design.

6.1 Example Application: Transparent Client-Side Caching for Key-Value Store Applications Using FPGAs

Key-value stores (KVS) are a critical component of the current data center infrastructure. They help address the extreme demand on data centers for high bandwidth, low latency access to large amounts of data. Due to their importance, many efforts have been made in prior work to improve their performance, which includes using FPGAs to offload some functionality. These efforts have always been focused on improving the performance of the key-value store itself and reducing the load on the server running the KVS. However, with more FPGAs being deployed in the data centers by many cloud service providers, some use models that were not practical sometime back are becoming more realistic. In this work, we explore one such use case where we attempt to cache key-value entries at the network interface of the client server. We propose an FPGA design that is capable of caching the KVS data transparently to

the KVS client application. The proposed solution is able to improve the application throughput while also reducing the network traffic generated by the KVS client. We also discuss the importance of and present our vision for an FPGA design supporting multiple tenants because the proposed solution targets the client servers that are typically shared by multiple clients.

6.1.1 Introduction

The demand on data centers for high bandwidth access to large amounts of data is growing due to various factors such as the rise of cloud services, the Internet of Things (IoT), and the general evolution of the data center landscape. Large content providers such as Facebook, Google, and Wikipedia. dedicate significant compute resources to key-value databases. Key value databases cache frequently accessed content, reducing the load on back-end databases while providing low-latency and high-throughput access to content. Key-value stores (KVS) are used for tasks such as storing user session data such as login information, storing configuration settings that applications need to access quickly, and numerous others.

Key-value stores, especially the ones geared towards caching small keys and values, pose challenges to data center servers due to the number of small network packets that must be processed. KVS clients generate a large amount of requests requiring the KVS server to process all the requests and generate the response packets at a high rate to ensure high throughput. This makes KVS services susceptible to bottlenecks introduced by the networking stack of the operating system. Due to this, many prior works such as [Choi et al., 2018, Chalamalasetti et al., 2013, Lavasani et al., 2013, Liang et al., 2016, Blott et al., 2013] have explored offloading key-value store functionality, fully or partially to FPGAs using custom hardware designs. These designs tightly couple the KVS functionality with the packet processing often with streaming architectures to provide higher throughput and lower latency compared to

a standard server.

While many of the prior works have focused on using FPGAs to accelerate the KVS implementation itself, this work focuses on the client interface. We propose a lightweight FPGA design for caching the KVS entries on a network-facing FPGA attached to the client server. The FPGA implements a smaller key-value store and handles requests sent out to the remote KVS server/s. If an entry corresponding to a request is already in the local KVS, the FPGA responds to the request without sending out the packet to the network. If not, the FPGA behaves as a regular NIC and sends the packet out. When a response packet is received, the value is cached in the local KVS while forwarding the response packet to the host. This allows the FPGA to service the next request for the same entry. In this design, the FPGA acts as a secondary network interface on the client-server and the caching happens fully transparent to the KVS client application. Our FPGA design presents a Virtio [Tsirkin and Huck, 2022] network device interface to the host machine. Virtio drivers are part of standard Linux distributions. Therefore, unlike most other FPGA-based designs there is no overhead of additional device drivers to communicate with the FPGA. From the point of view of the host operating system, the FPGA appears as a regular network interface. The only configuration step necessary is creating routing table entries to ensure that the packets intended for the remote KVS server are directed to the FPGA network interface instead of the primary NIC of the server.

We envision cloud applications that frequently access content from backend databases and use key-value stores as an intermediate caching layer benefiting from the proposed solution. For instance, consider a front-end web server running on a cloud server node. It needs to access the backend databases to fulfill requests from the users. In such a scenario, it is typical to have a key-value database such as Memcached [Memcached, 2024] running on a different set of nodes to cache the fre-

quently accessed content. The proposed solution can be deployed on the front-end server if it is equipped with an FPGA. This is becoming a realistic requirement as many major cloud service providers have started deploying FPGAs in their data centers [Microsoft, 2024b, Caulfield et al., 2016, Firestone et al., 2018, Amazon Web Services, 2024, Alibaba Cloud, 2018, Nimbix, 2024]. With caching at the network interface of the front-end server, the application should experience higher throughput due to a portion of requests being serviced without requesting data from the node running the KVS. Additionally, this will also reduce the network traffic on the data center network. In this work, we present our preliminary implementation with a single network interface and a KVS on the FPGA. However, this work can be extended to implement multiple PCIe functions that could be assigned to different guest VMs on a server. The individual PCIe functions and the associated controllers on the FPGA could serve different applications and provide performance isolation and quality of service guarantees for different clients.

6.1.2 Method

Scope for the Proposed Solution

We propose an FPGA design to perform caching at the network interface for key-value store applications. This solution targets servers running client applications that need to access data from backend servers and rely on key-value databases that run on dedicated nodes to cache frequently requested data and reduce the load on the backend servers. The target server should be equipped with a network-facing FPGA. The FPGA is able to act as a secondary network interface card for the server. Figure 6·1 depicts the general deployment context targeted by the proposed solution. Here the frontend server (server ①) is running some application that serves the requests from the users. The data required to service the user requests are stored in the database on server ②. Server ③ runs a key-value store that caches the

frequently accessed data in order to reduce the load on the server (2).

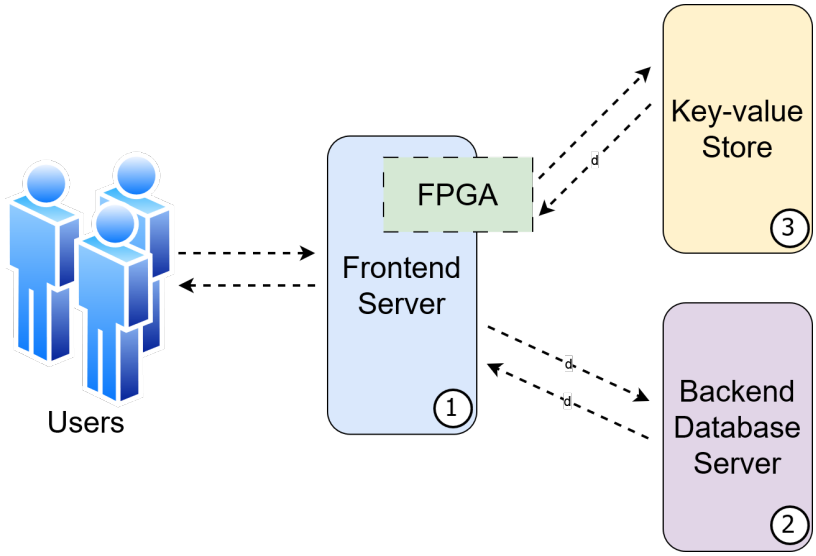


Figure 6·1: Target deployment architecture (Adapted from [Choi et al., 2018])

With our proposed solution, all packets between servers (1) and (3) are sent through the network-facing FPGA on server (1). The FPGA implements a smaller key-value store and the network interface functionality. The architecture of the FPGA design is described later in this section. The proposed FPGA design includes a host interface compliant with the Virtio specification [Tsirkin and Huck, 2022]. This allows the host OS to use in-kernel Virtio device drivers to communicate with the FPGA. Because our concrete implementation is a Virtio network device, the FPGA appears as a regular network interface card for the host OS. Because our current implementation does not implement any additional functionality such as checksum calculations or TCP segmentation offload, it relies on the host operating system's network stack and the device driver to provide fully formed Ethernet frames to the FPGA. However, because Virtio specification supports feature negotiation between the device and the driver, future implementations can implement additional network stack functionality and enable offloading more work from the CPU to the FPGA. Any such additions do

not require any changes to the current use model where the KVS client is oblivious to the existence of the FPGA or the caching at the network interface.

After a packet is copied to the FPGA memory from the host memory using DMA, the KVS module reads the relevant fields from the packet to identify the type of request and the key. At present, our implementation only supports `Get` and `Set` requests in the Memcached protocol. A `Get` request triggers a KVS lookup. If the requested key is available in the local KVS, the FPGA generates a response ethernet frame and moves it to the appropriate buffer in the host memory. If the key is not found, the request packet is sent out into the network using the FPGA's network interface. When the response packet is received, the KVS module inserts the value into its memory while the DMA controller moves the packet to the host memory. The FPGA's behavior for a `Set` request is configurable. The two modes supported are: (i) `Set` requests trigger a KVS lookup; entry is updated for a hit; the request packet is sent out to the KVS server (ii) `Set` requests are never cached; If the corresponding entry is in the local cache, it is invalidated; Packet is sent to the remote KVS server. The behavior should be selected based on the kind of data cached and the consistency model enforced by the remote key-value store. For all other request types, the local KVS is bypassed and the packet is sent to the network interface. Any local copies of the entry targeted by the unsupported request are invalidated. Essentially, the FPGA behaves as if there is no local cache in place for all unsupported requests.

Architecture

Figure 6·2 provides an overview of the FPGA design. The main components of the design are the Key-value store, PCIe subsystem, and the network subsystem. Additionally, there are packet buffers to store RX and TX packets, and a central controller is responsible for managing the other components.

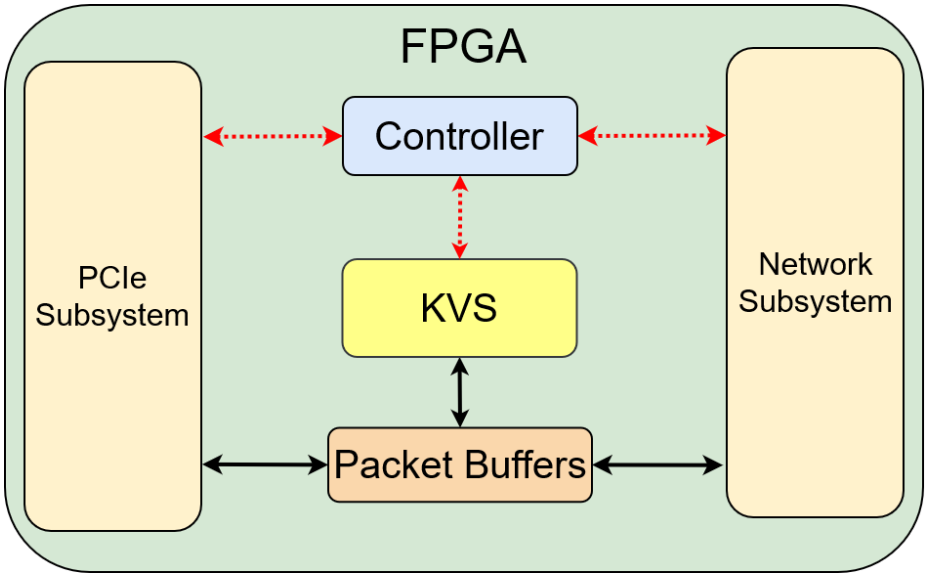


Figure 6·2: KVS Design Overview

Key-value Store: The key-value store implementation complies with the Memcached binary protocol [Memcached, 2017]. A simple CRC16 hash is used as the hash function for the KVS. This allows us to implement a KVS with up to 2^{16} entries. Keys and values are stored in FPGA block RAMs (BRAM) to simplify the design. However, the design could be modified to use off-chip DRAM to store the values and BRAMs to store keys and pointers to the values in DRAM. The KVS is implemented as a direct mapped cache. Therefore, no replacement policy is implemented. A single FSM handles the orchestration of all computations, moving data to and from packet buffers, reading/writing keys and values from/to BRAMs, and communication with the controller.

PCIe Subsystem: This design uses the PCIe subsystem presented in Chapter 4 to implement a host-FPGA communication interface where the host OS views the FPGA as a Virtio network device. This removes the overhead of designing and implementing custom device drivers for the FPGA developer and the overhead of setting up and maintaining the drivers for a system administrator. Since the FPGA appears

as a regular network interface card, no changes are necessary for the KVS client application. Packets generated by the KVS client application can be directed to the FPGA by setting up the proper routing table entries.

We can use a soft-core processor as the controller to reduce the number of custom modules that need to be implemented. Figure 6·3 depicts the KVS design with a RISC-V core as the controller. In this architecture, the 'Memory' module acts as a shared memory for all the components. The PCIe, network, and KVS modules have direct access to the memory. The configuration registers of all three modules are memory-mapped to the address space of the RISC-V processor allowing the processor to control the other modules. At initialization, the program executing on the processor allocates packet buffers in the memory and provides the pointers to the buffers to the other modules.

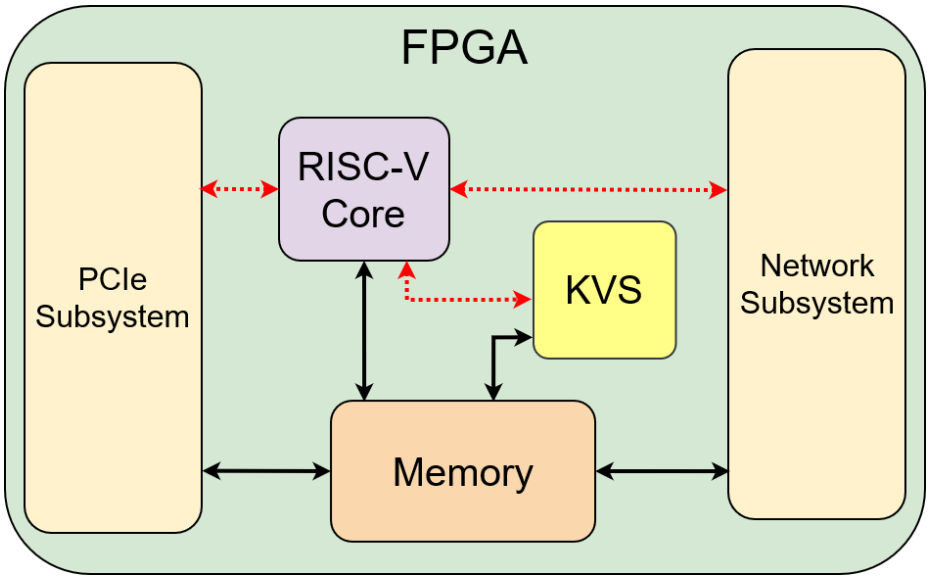


Figure 6·3: KVS Design with RISC-V Processor as the Controller

Extension to Multiple client Applications/VMs

Since Virtio drivers are intended for guest operating systems and the drivers do not make any distinction between virtual and physical devices as long as the correct

interface is implemented, it is possible for guest VMs to also access the FPGA without additional device drivers. We have tested our current implementation with a single PCIe function and a single VM by using PCIe passthrough to assign the FPGA to the VM. We expect to extend this approach to multiple VMs using multiple PCIe functions and the Single Root I/O Virtualization (SR-IOV) technique. This suits the target deployments we have considered for this solution, which are cloud nodes running applications that are key-value store clients. Typically such applications run inside virtual machines and share the server resources with other applications.

If there are multiple KVS client programs executing inside different VMs on the same host machine, it is important to provide each of them with a simple but secure mechanism to access the FPGA. If the different clients are accessing different key-value databases, the corresponding PCIe functions implemented on the FPGA could have their own KVS controllers and memory. Even if the different clients are accessing the same database, each could be assigned their own cache on the FPGA in order to provide additional security, performance isolation, and quality of service guarantees.

Figure 6·4 depicts our vision for a multi-tenant deployment where multiple KVS client programs are running inside individual VMs on the same server. The PCIe interface of the FPGA presents multiple PCIe functions (physical or virtual) to the host OS, which are assigned to individual VMs. VMs 2, 3, and 4 are running KVS clients and therefore are connected to the FPGA. VM 1 could be running a non-KVS program and therefore is not connected. Within the FPGA, functions 3, and 4 share the same KVS controller. However, function 2, which is connected to VM 2 uses its own KVS controller. This could be due to VM 2 accessing a different KVS from the other two VMs or due to security or performance guarantees required by the program running in VM 2. The KVS controllers implemented on the FPGA share the external interfaces such as PCIe and Ethernet.

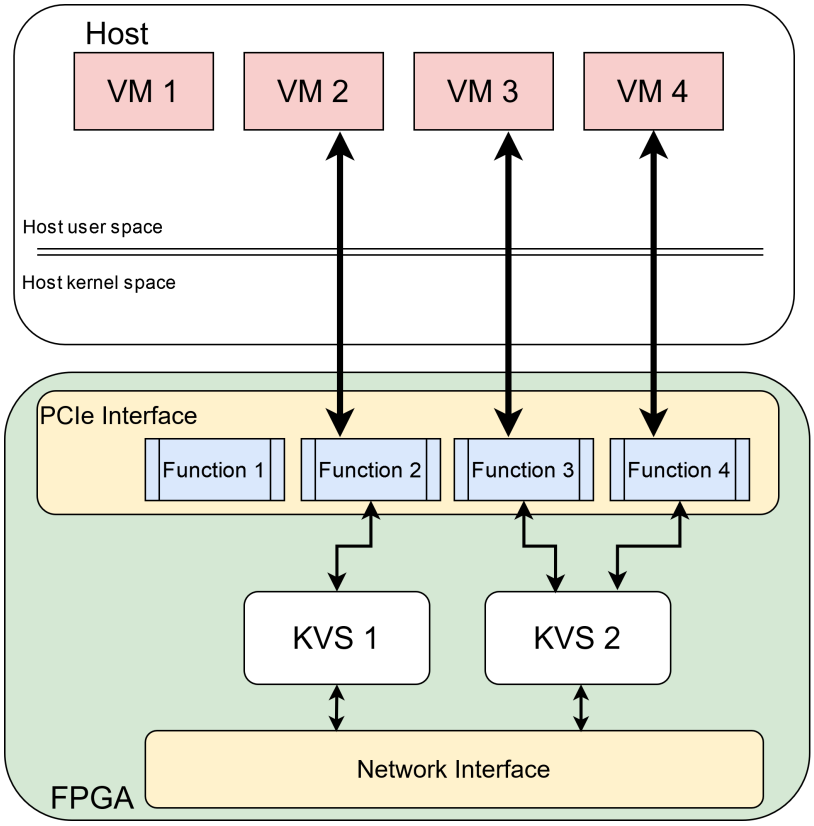


Figure 6·4: Example Multi-tenant Deployment

6.2 Building a System Using DISL

One of the main motivations behind using a system generator such as DISL is to simplify and speed up the implementation process. An hOS generator allows a user to focus on application logic by automatically generating the infrastructure logic based on the application requirements specified by a user. Therefore, a good hOS generator design should result in a straightforward process for defining and implementing a system. This idea can be broken down into the characteristics of the hOS generator such as listed below.

- The methods, and syntax used to describe a system.
- The amount of information a user is expected to provide to the system generator.

- The level of hardware design expertise required.
- The level of understanding the user should have of the inner workings of the system generator to effectively use it.

A good system generator design should use easy to understand methods, syntax, and constructs to define a system. A user should not have to spend a lot of time and effort specifying and generating a system such that the overhead of specifying a system outweighs the time saved by using the system generator instead of implementing the system manually. There should also be a balance between the simplicity and expressiveness of the system specification methods. For instance, a graphical user interface (GUI) may be far easier to familiarize with compared to a text-based system specification. However, the GUI may lack the expressiveness necessary to specify certain aspects of a design. It is possible for a system generator to use different methods/syntax/use models to achieve simpler versus complex tasks. For example, consider our previous example of GUI vs text-based system specification. The two methods can be combined to provide a simpler use model for straightforward implementations and a more expressive use model to implement complex designs.

A system generator uses a system specification provided by the user. A good system generator design should be able to build a working system with the minimum amount of information from the user. This benefits both the users with a lower level of hardware design expertise and users who wish to quickly generate a working design and then optimize it if necessary. While a user should be allowed to provide additional information to fine-tune the design, it should not be the baseline use model. It should be possible to generate a working design without understanding the intricate details of the components used in the design. A system generator can achieve this by using default configurations and parameter values for the system components. The maximum amount of information a system generator can expect from a user

is the same as if the user provides HDL source files, configuration information and parameter values for the system components not included in the source files and all the design constraints. A good system generator design should never require more information than this from the user regarding the system to be generated and ideally require a lot less to generate a design.

A good system generator design should be usable by users with different levels of hardware design experience. The use model could be different for different users. For users who lack hardware design expertise, the hOS generator can decide most of the configuration parameters for the system components. Default values and example designs can be useful in such a use model. However, an experienced hardware designer should have access to the hardware parameters of the components to optimize them to match the application requirements better.

From a usability standpoint, a user should be able to use a system generator with zero knowledge of how the generator is implemented and how it functions. However, an ideal design should also allow an expert user to easily change the behavior of the tool to either change the current system generation mechanics or add new capabilities. By examining the points discussed above, it is clear that the ease of implementing a system with a system generator cannot be easily quantified. Therefore, we provide a qualitative evaluation of DISL.

DISL uses configuration files written in TOML [Preston-Werner, Tom and Gedam, Pradyun et al., 2023] which is a minimal configuration file format; to describe systems, library components, and board specifications. TOML syntax is designed to map information to hash tables. Much of the DISL use model and the level of difficulty in using DISL revolves around how a user interacts with the configuration files. We envision DISL being used by users with a wide range of experience in hardware design. The use model changes with the level of expertise and therefore, the effort to use DISL

is discussed relative to different levels of expertise.

User has no experience in hardware design: This is an extreme scenario where the user has no experience in hardware design and does not have an understanding of the system components necessary to implement a functioning FPGA design. For such a user, the motivation behind using an FPGA could be something other than performance. For example, security could be one such motivating factor where the user needs to run a security-critical application on an external device protected from the host OS and other programs executing on the host.

As the user is not proficient in implementing a hardware design, they could use a soft processor implemented on the FPGA to execute a program the user has written. In this scenario, the user can use an example design available in the DISL repository to generate a system and implement their application in software to be executed on a soft processor core. An example system configuration for such a use case is shown in figure 6·5. The example system implements PCIe, network, DRAM, and UART interfaces alongside a RISC-V core. The different interfaces are memory-mapped. Note that the 'Interconnect' block represents the automatically generated interconnect logic which is part of the top module and not a separate module.

One drawback of this use model is that the user might have to use a system configuration with components unnecessary to their application. While the use model is simple, the resulting system design may not be optimal. However, we are assuming a scenario where the user has no expertise in hardware design and does not understand the system components necessary to make the FPGA design work. The user cannot provide system requirements without understanding them. Yet, the user is no worse off than using a fixed FPGA shell implementation as those also use a fixed amount of FPGA resources regardless of the specific application requirements.

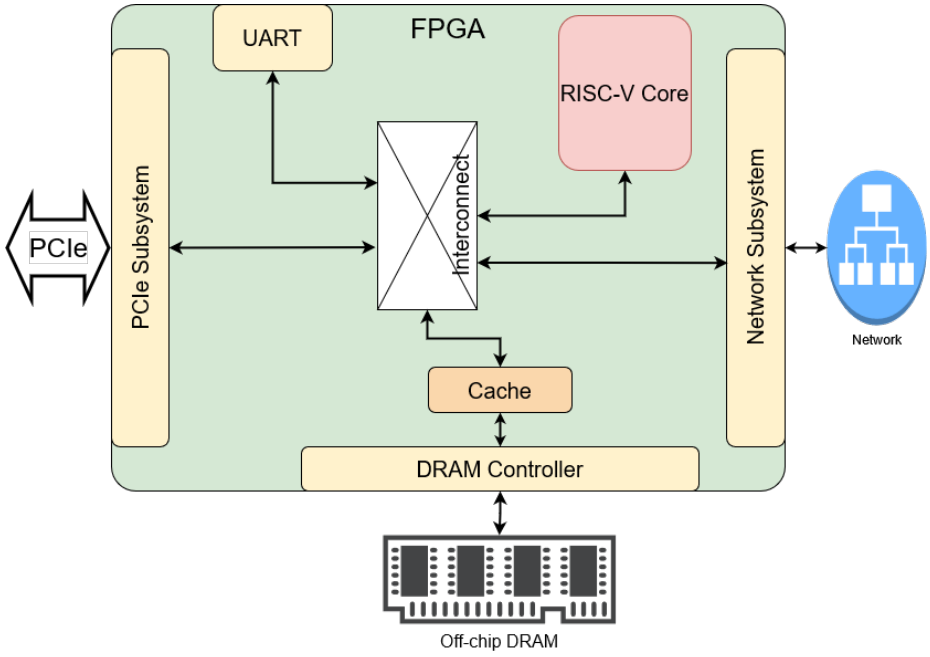


Figure 6·5: An example system configuration with a softcore processor

User understands the system components required by the application: If a user has an understanding of the different components required by their application can select an example design that only includes the required components. If such an example is not available, the user can start with an example such as the one shown in Figure 6·5 and remove the unnecessary components from the system definition such that the resulting system only includes the required components. For instance, let us consider an application that does not require off-chip DRAM. The application only uses on-chip BRAM for memory. In this case, the DRAM controller can be removed from the system configuration by removing the entry corresponding to the DRAM controller from the `[INSTANTIATIONS]` dictionary in the `system.tml` file. Next, the connections between the DRAM controller and the rest of the components should be removed. This is done by removing the entries corresponding to the DRAM controller from the `[INTERCONNET]` dictionary. Because the connectivity is defined on a per-interface basis, unnecessary connections can be removed without affecting the rest of

the connectivity. However, the amount of changes to the system specification depends on the exact component to be removed. Removing the DRAM controller from the system shown in Figure 6·5 is trivial because it is only connected to the `'cache'` module. However, removing a module directly connected to the RISC-V core and memory mapped may also require updating the address map of the processor instance.

The text-based detailed system specifications of DISL have some drawbacks in this scenario. While this scheme provides greater control over the implemented system to an experienced user, it may not be ideal for a less experienced user who wishes to make minor modifications to a system. This can be improved with another abstraction layer used to generate the text-based system descriptions. In Chapter 7, we discuss this idea further as part of future research directions.

User implements application logic: There could be users with sufficient hardware design experience to implement their application logic as a hardware module but lack the experience to implement external interfaces such as PCIe. It could also be the case that the user wants to implement a working design quickly without spending time implementing the infrastructure logic required for the application logic to function.

A user can start with an example design similar to the last two use models, but add their hardware module to DISL to generate a system with application and infrastructure logic. The first step is adding a description of the user module to `modules.tml` file. Next, the instances of the new module are added to the `[INSTANTIATIONS]` dictionary and the connectivity is specified in the `[INTERCONNET]` dictionary in `system.tml`. This use model requires the user to understand the system components and the connectivity between those and the application logic. If the new hardware module uses interface types not already specified in `definitions.tml` the interface definitions also need to be added to DISL.

In addition to the steps above, a user can optionally add parameter mapping functions to the system generator script if the user module has complex hardware parameters that are better abstracted away with simpler user-facing parameters. This is important if the user is expecting to reuse the module or the user is implementing components to be used by others.

Good system component designs are important for this use model because the user is required to specify the connectivity between system components and their hardware modules. The system components should be designed to present consistent and easy to understand interfaces to the rest of the system. Ideally, the interface should be designed such that the user does not have to understand the inner workings of the component. For example, the PCIe subsystem discussed in Chapter 4 can present one of two interfaces to the rest of the system. The first is an AXI memory-mapped interface which is a straightforward handshake-based interface widely used in IP cores and most hardware designers are familiar with. The second is a Virtqueue-based interface which software developers may find easier to understand and use. Neither of these interfaces requires the user to understand the internal details of the PCIe subsystem, such as the PCIe IP core used, how the DMA engine is programmed, or how the Virtio device drivers on the host machine recognize the FPGA as a Virtio device. From the point of view of the user logic on the FPGA, communicating with the host machine takes the form of either sending and receiving data over an AXI interface or using a Virtqueue abstraction which involves buffers and data structures in FPGA memory.

Porting a design to a new device: When using DISL, porting a design is a straightforward process if the support for the new device is already implemented. This means the device-specific components for the new device are implemented by a previous user and are added to the DISL repository. In this case, a user can take

one of two approaches. The first is to start with the example design for the new device and add the user logic to the system specification. The second is to start with the current system specification (for the old device) and replace the device-specific components in `system.tml` with the components for the new device. Which approach works best depends on how complex the connectivity between the application logic and the rest of the system is. If the application logic is mostly isolated and has few interfaces connected to the rest of the system, the first approach works best and vice versa.

User implements new system components: In all the use models discussed so far, the user is using system components implemented by someone else and added to DISL. A user can also add new system components to DISL if not already available or if the current version needs to be improved. If the new component is device-agnostic the process is no different from adding application logic to DISL. However, if it contains device-specific elements, `board.tml` file should be updated with any information and methods necessary to generate and compile the device-specific elements. These could include design constraints, and commands to generate IP cores. A description of the highest-level module of the new element is added to `modules.tml`.

Adding support for a new device: The next level of the different use models for DISL is adding support for new development boards. Adding a new device to DISL requires creating a directory for the new device in `/fpga/boards/` directory and adding a board specification (`board.tml`) and any device-specific sources such as source files, and scripts related to the device-specific components to the new directory. The steps are described further in Section 3.3.1.

At this level, the user is expected to have a solid understanding of the device-specific components and how to implement them. However, because DISL does not

have a minimum set of capabilities specified, adding support for a new device can be done incrementally. One user could add only the components necessary for their application and a later user can reuse these components and add any missing components to improve device support.

Modify DISL: At this level, the user is not simply using DISL to generate designs, but modifies how DISL generates a system and adds more functionality to DISL. The system generator (`/fpga/system_builder/build.py`) and configuration (`configure.py`) scripts are updated to add new functionality to DISL. For instance, the configure script could be updated to add support for more FPGA toolflows. Changing how the RTL is generated requires modifying the system generator script. An example of a new feature is including some form of memory protection logic in the generated RTL. This could be used in a use case where multiple application kernels from mutually untrusting users are sharing the FPGA.

6.2.1 Usability Evaluation

There is a range of use models for DISL that can be used by users with different levels of hardware design experience as described previously. At the lower end, DISL offers simple use models that mostly depend on example designs. The user interactions with the configuration files are minimal. Therefore, the effort to generate a design using DISL is less. DISL allows a less experienced user to quickly generate a working design without a deep understanding of the system components or the inner workings of DISL.

At the higher end, DISL still allows more experienced users to implement complex functionality, improve available components, add new system components, and add support for new devices. These require increasingly higher levels of understanding of hardware design, using IP cores, implementing external I/O interfaces, and how

DISL functions. These also require higher effort from the user. However, we believe the increased difficulty in using DISL is proportional to the nature of the outcome the user aims to achieve. Simply using DISL to generate a system with already available components for a board that is already supported by DISL requires minimum effort. At the other end of the spectrum, changing the behavior of DISL itself requires significantly higher effort.

The scripting and programming languages used in implementing and using DISL increase its usability. DISL is implemented in Python which is simple and widely used. Adding parameter mapping functions or adding new functionality to DISL only requires the user to be familiar with Python. Most of DISL functionality revolves around the configuration files. They use TOML [Preston-Werner, Tom and Gedam, Pradyun et al., 2023] which is a minimal configuration file format with obvious syntax. The syntax is easy to read, understand, and modify.

Something a new user might find challenging is keeping track of exactly which configuration files need to be modified for different tasks. Using DISL to generate a system with existing components only requires a new `system.tml` file. However, adding new components to DISL may require the user to modify other configuration files such as `modules.tml` and `definitions.tml`. Adding support for new devices involves yet another configuration file, `board.tml`.

Another aspect of DISL that might be challenging to a new user is understanding the reasoning behind certain system definition rules. For instance, the interconnect is specified using three different dictionaries `[STATIC]`, `[DYNAMIC]`, and `[OVERRIDES]`. This is done due to two main reasons. DISL generates the RTL source for the interconnect that connects the system components. Only some connections require arbitration logic. Therefore, the `[STATIC]` and `[DYNAMIC]` dictionaries separate the connections that do and do not require arbitration logic. The `[OVERRIDES]` dictio-

nary provides more flexibility in specifying the interconnect by allowing the user to override the connectivity of individual signals that are part of an interface regardless of the connectivity specified for the full interface. Another example of a DISL system specification rule that may confuse a user is how the interfaces are broken down into individual handshakes. This is done to support conversions between different protocols.

DISL configuration files use a highly detailed approach to describing components and systems. While this is necessary to gather enough information to generate the 'top' module of the design and this provides greater control over the design, it could be a drawback for some users. However, the amount of details a user needs to provide can be further reduced by using another abstraction to generate the configuration files. At present a user is required to specify the application requirements in terms of system components and their connectivity. Another layer of abstraction could allow a user to specify requirements at a higher level with fewer details.

6.2.2 Building the KVS Design Using DISL

At the time of implementing this example design, we did not have a fully functioning network subsystem. Therefore, we have implemented a design without the network connectivity as shown in Figure 6·6. This design still allows us to evaluate the performance of the key-value store in conjunction with the Virtio PCIe interface. The results we present later in this section are based on performance models where the latency for communication over the network is modeled based on the communication latency over the default NIC on the host machine. We have also included a UART module in the implemented design to make it easier to debug the design. The RISC-V core can print messages over the UART interface that we can use to debug the design.

The first step of generating the design is adding the key-value store module to the DISL component library. This is done by adding the module description to

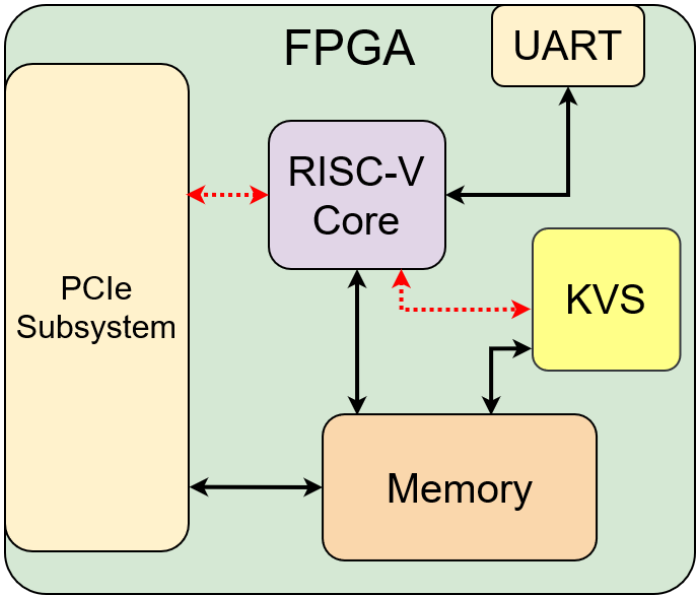


Figure 6·6: KVS Design Implemented with DISL

'`/fpga/common/config/modules.tml`'. It is not necessary to update the other configuration files because the KVS module only contains generic logic and only uses interfaces already defined in DISL. Listing 6.1 shows the entry in `modules.tml` describing the KVS module. The next step is specifying the system to be generated. The full system description for the key-value store design is provided in Appendix A.

```
...
[kvs]
  TYPES = ["SIMPLE"]

  PARAMETERS = ["LISTEN_PORT", "HASH_WIDTH", "KEY_WIDTH", "VALUE_WIDTH", "
    VALUE_SIZE"]

  [kvs.REQUIREMENTS]
    INTERFACES = ["clk", "rst", "s_axil", "m_axil"]
    [kvs.REQUIREMENTS.INCLUDES]
      COMMON = ["kvs.sv", "bram_simple.v", "crc16.v"]
      BOARD  = []

  [kvs.ENCODINGS]

  [kvs.INTERFACES.clk]
      TYPE = "CLOCK"
      DIRECTION = "SINK"
  [kvs.INTERFACES.rst]
      TYPE = "GENERAL"
```

```
20          WIDTH = 1
21          DIRECTION = "SINK"
22      [kvs.INTERFACES.s_axil]
23          TYPE = "AXIL"
24          DIRECTION = "SINK"
25          CLOCK = "clk"
26          DATA_WIDTH = 32
27          ADDRESS_WIDTH = 32
28          MASK_WIDTH = 4
29      [kvs.INTERFACES.m_axil]
30          TYPE = "AXIL"
31          DIRECTION = "SOURCE"
32          CLOCK = "clk"
33          DATA_WIDTH = 32
34          ADDRESS_WIDTH = 32
35          MASK_WIDTH = 4
36  ...
```

Listing 6.1: Adding the Key-value Store Module to modules.tml

6.3 Matching the Exact Application Requirements

One of the main drawbacks of fixed FPGA shells is the fixed resource usage regardless of the capabilities required by a given application. An hOS generator can remedy this by tailoring the generator system to the exact application requirements. In this section, we explore how much the system requirements differ among different applications and how DISL can generate system designs consisting only of the features requested by a user via a system specification.

Table 6.1 presents a taxonomy of different application classes and deployment contexts for FPGAs and the components likely to be included in system designs for each of the use cases. Note that this is not a complete taxonomy of all possible use cases for FPGAs. Instead, this covers a select few use cases to highlight how the set of required system components differ significantly across different deployment contexts and applications.

A good system generator design should allow the user to individually select the system components to match the application requirements. This improves the overall

	Coprocessor	Cluster	SmartNIC	Network Storage	IoT (Data logging)	IoT (Control)
PCIe	✓	✓	✓			
Wired Network		✓	✓	✓		
Wireless Network					✓	✓
DRAM	✓	✓		✓	✓	
NVMe				✓	✓	
GPIO						✓
Sensors					✓	
UART	✓	✓	✓			
Soft cores	✓	✓	✓	✓	✓	✓

Table 6.1: Example Taxonomy of FPGA Use Cases

resource usage compared to a fixed FPGA shell as there are no unnecessary components in the design. This indirectly results in energy efficiency as well because the FPGA resources can be used to implement more compute units that perform useful computations instead of implementing unnecessary infrastructure logic.

DISL generates systems using a component library. A user is required to specify each component to be included in a system design through a system specification. Therefore, assuming the user provided an accurate system specification, only the components necessary for a given application are going to be included in the generated system. Depending on the resource availability of the target device, selecting the correct system components can have a significant effect on how much resources are available to implement the application logic after implementing the system components. Going one step beyond this, by applying the design strategy discussed in Chapter 3, the system components themselves can be built out of discrete subcomponents that can be individually selected/de-selected as necessary. So far, this strategy was presented with a focus on the design portability aspect. However, an additional benefit of the modular approach to component design is that it allows an advanced

user to add/remove/replace sub-components to implement highly optimized designs targeting strict resource constraints or performance targets.

For example, consider the PCIe subsystem presented in Chapter 4. The PCIe subsystem design includes a modified version of the vendor-provided PCIe IP core in the device-specific (`Board`) partition and a controller module in the `Generic` code partition. The controller implements multiple RX and TX queues and arbitrates the requests from different queues to access the single interface exposed by the PCIe IP core. The controller also presents different interfaces to the user logic based on user specifications. However, these added capabilities require additional resources on the FPGA. There are scenarios a user may opt to get rid of the `Generic` portion of the PCIe subsystem and directly connect user logic to the PCIe IP core. One reason could be strict resource constraints. Another could be that the user requires a highly optimized host-FPGA interface for a single accelerator that does not benefit from having multiple communication queues and the added latency from the controller is unacceptable. Due to the layered component design strategy, DISL can support such use cases where a user opts to remove/replace sub-components of a subsystem in the hOS.

At present, DISL does not support hierarchical system specifications. This means the user can only specify the components and connectivity in the top module of the design hierarchy. Therefore, the components that include sub-components that can be selected/deselected cannot be described using configuration files. Instead, the top module of the component has to be parameterized to only instantiate certain components depending on values of configuration parameters. If not, the sub-components can be instantiated individually by adding them individually to the `[INSTANTIATIONS]` dictionary in `system.tml`.

Based on the discussion above, we can conclude that DISL provides a high de-

gree of freedom in terms of tailoring the generated system to the exact application requirements resulting in lower resource overhead. At this point, we shift our focus back to the example design discussed previously. In the next section, we discuss how the previously discussed design is ported to a new device.

6.4 Porting Designs to New Devices

DISL enables a design to be ported to different devices by implementing only the device-specific components used by the generated system design. In this section, we port the design shown in Figure 6·6 from the smaller Artix 7 device to the larger Ultrascale+ device. Among the components used in the design in Figure 6·6, the PCIe subsystem is the only component that includes device-specific submodules. (The UART module requires the proper pin placement constraints which are part of the board description in the `board.tml` file. However, the module is completely generic). Because we ported the PCIe subsystem to the Ultrascale+ device and added board support for the CVP13 development board, we can port the design by only creating a new system description that uses the correct components.

In this implementation, we did not use a wrapper module for the PCIe subsystem. Instead, the XDMA IP core, `pcie_controller` module, and the other modules that are part of the PCIe subsystem were instantiated directly in the top module. This results in many signal and instance name changes between the two system descriptions for the two target devices. However, if we ignore the signal and instance name changes, the difference between the two system descriptions is only 10 lines. Considering that the full `system.tml` file is 470 lines long, the changes required to port the design to a new device (board support already added to DISL) are negligible.

6.4.1 Scaling Designs

Apart from easily porting designs across devices, DISL also simplifies scaling designs to match the resource availability of different devices given the components are properly parameterized. Listing 6.2 shows the entry in the `system.tml` file corresponding to the KVS module instance. The parameters `'HASH_WIDTH'`, `'KEY_WIDTH'`, and `'VALUE_WIDTH'` determine the amount of memory (BRAM) used by the key-value store module. The depth of the memory is determined by `'HASH_WIDTH'`. The other two parameters determine the size of a memory line for the memory blocks storing the keys and the values respectively.

We have kept the `'KEY_WIDTH'` fixed at 32 bits. The configuration shown in Listing 6.2 is the configuration that uses the largest amount of BRAM that would still fit in the Artix 7 device. However, the Ultrascale+ device has more than 7 times the number of BRAM tiles as the Artix device. Therefore when porting the design, we could also utilize the additional resource by making the key-value store larger. This could mean either supporting larger values or more entries in the key-value store. These alternative configurations can be generated by changing the values assigned to `'HASH_WIDTH'` and `'VALUE_WIDTH'` parameters in `system.tml` and rerunning DISL to regenerate the design with the updated parameter values.

Scaling a design with more processing elements (PE) is also not significantly more difficult to achieve. This requires the user to add the new processing element instances to the `system.tml` file and regenerate the design. However, one limitation of the current system definition syntax used by DISL is the inability to instantiate modules conditionally or inside a loop. Therefore, scaling the number of PEs requires adding new instances one by one to the `[INSTANTIATIONS]` directory in `system.tml`. The component design can provide a workaround by implementing a wrapper module with a `'generate for'` loop to instantiate the PEs and a parameter to set the number of

instances.

```
[INSTANTIATIONS.kvs_inst]
  MODULE = "kvs"
  PARAMETERS.LISTEN_PORT = 44000
  PARAMETERS.HASH_WIDTH = 16
  PARAMETERS.KEY_WIDTH = 32
  PARAMETERS.VALUE_WIDTH = 128
```

Listing 6.2: KVS module instance in system.tml

Overall, DISL provides simple and easy to use methods to port designs across devices and scale the designs to match the resources available on the new device. The component design strategy is the driving force behind the improved design portability.

6.5 Evaluation of the Key-Value Store Design

In this section, we evaluate our key-value store design [Bandara et al., 2024a]. We model the application throughput and the network bandwidth using the timing measurements for data movement between the host and the FPGA and the network latency to reach the server running the remote key-value store.

The purpose of allocating compute resources to run key-value stores is to provide fast access to data frequently accessed and reduce the load on backend databases. However, since traditional servers a prone to performance bottlenecks when running key-value stores, many FPGA-based solutions have been proposed to improve the throughput and latency of the key-value stores. These approaches are typically evaluated based on the improvement in application throughput. Therefore, we also evaluate our proposed solution based on application throughput.

6.5.1 Experimental Setup

We use our implementation on the Bittware CVP13 [Bittware, 2024] development board that uses a Xilinx Ultrascale+ FPGA (part number: xcvu13p-figd2104-2-e) for this evaluation. The machine hosting the FPGA and running the KVS client

program and the server running the key-value store program are both connected to the same 1 Gbps local network. We consider the KVS client performance without the FPGA-based solution as the baseline for our analysis. For time measurements, the test applications use the `clock_gettime()` function with the `CLOCK_MONOTONIC` option.

Since the real KVS workloads are dominated by read requests [Atikoglu et al., 2012], we focus our analysis on the performance improvement for Memcached `Get` requests. Higher hit rates on the local KVS cache correspond to better client application performance since fetching a value from the FPGA takes considerably less time compared to fetching the same value from a remote server running the KVS. We assume that the KVS client application's performance is only limited by accessing the key-value store on a remote node.

6.5.2 Results

In this section, we model the performance improvement achieved by the proposed solution. The model is based on two types of latency measurements made using the setup described in Section 6.5.1.

1. Time taken for the client program to issue a `Get` request to the key-value store running on a different machine on the same network, and receive the response with the value. This is considered to be the baseline performance when there is no FPGA KVS cache implementation in place.

2. Time for the client program to receive the response when the KVS entry requested is cached in the FPGA.

Figure 6·8 shows the effective read throughput for the client program for different sizes for the value field ranging from 16 Bytes to 512 Bytes versus different cache hit rates for the KVS cache in the FPGA. Performance for the 0% hit rate is the

same as the performance without the FPGA cache in place. The highest performance improvement is seen for the smallest values. This can be explained by the higher overhead of network packets for smaller values. For the smallest sizes, the overhead of the protocol headers alone is closer to the size of the actual payload. The performance benefits decrease with the size of the value. However, the negative slope of the (light blue) line representing 512 Byte values cannot be explained only by the decreasing overhead of the network packets compared to smaller value sizes. The other factor contributing to the poor performance is that our FPGA design is still not optimized to handle larger values efficiently. With an optimized design that can handle larger values better, the proposed solution should result in improved performance for a wider range of payload sizes.

Figure 6·9 shows the network bandwidth versus the cache hit rates. Please note that this is the network bandwidth perceived by the client application and not the actual bandwidth utilized. Because the application is unaware of the caching at the network interface, from the point of view of the application, it is experiencing higher network bandwidth utilization. For the no-cache scenario (0% hit rate), the application only reaches around 7% of the available network bandwidth of 1 Gbps when fetching 16 Byte values from the KVS. However, with a 100% hit rate, it could go up to around 13% which is almost double the bandwidth for the no-cache case. The hidden benefit of this is that the actual number of packets transmitted over the network is reduced lowering the actual network traffic. The reason for the poor performance of the 512 Byte value scenario is the same as explained above.

6.6 Optimizing System Components

Our evaluation of the key-value store design showed that it does not perform well for larger values. In this section, we investigate the reason behind poor performance and

attempt to modify the design to obtain better performance. In doing so, we evaluate the steps involved in modifying the system components used in DISL.

The performance of our design suffers when handling larger values. Therefore, we can infer that the added overhead is related to the larger payload. Remember that we have built our performance model based on two timing measurements. Cache hits are modeled using the latency measured using our FPGA design. This measurement represents the latency for the KVS client application to send a packet, the packet to be moved to the FPGA over the Virtio interface, and the KVS module on the FPGA to look up the key and respond with the value if it is cached. Because we did not have a functioning network subsystem at the time, the cache misses were modeled using a different measurement. That is the latency for the KVS client application to send a request to the server running the key-value store over the default NIC of the machine and to get a response from the remote server.

The performance for the 512-byte values in Figures 6·8 and 6·9 does not make sense because the graph suggests that the performance decreases with higher hit rates. This means that fetching a value from the FPGA is more expensive than fetching one over the network. In Chapter 5, we have demonstrated that Virtio drivers provide comparable or better performance to regular device drivers. Therefore, it is most likely that the added latency when moving larger payloads between the host and the FPGA is related to the actual PCIe transactions rather than the software stack. Considering all the factors above, we can conclude that the poor performance for larger values is at least partially due to the design not fully utilizing the available PCIe bandwidth when moving data between the host and the FPGA.

This is in fact true because we did not account for some of the improved capabilities of the new device when porting the design from the Artix 7 device to the Ultrascale+ device. In Section 6.4, we only focused on porting the design to a new device with

minimal changes to the components and the system specification. However, it may be necessary to further optimize a design to achieve the highest performance. We do not consider this to be a drawback of DISL. Rather it is part of the DISL use model which allows a user to quickly port a working design to a new device and optimize it incrementally if necessary. In this case, a user can port the design to a different FPGA with minimal changes to the system specification. If the performance of the resultant implementation is satisfactory, the user can use that. However, if the implementation cannot meet the performance targets, the user can then optimize the design incrementally.

Remember the differences between the two FPGA development boards we listed in Section 5.6. The development board that uses an Artix 7 device only supports two PCIe 2.0 lanes. On the FPGA side, the XDMA IP is implemented with a 128-bit data bus operating at 125MHz. In contrast, the CVP13 development board which uses an Ultrascale+ FPGA supports a 16-lane PCIe 3.0 interface. To use the full bandwidth available for this interface (8GT/s per lane), the XDMA IP should be configured with a 512-bit interface operating at 250MHz. While our design for the CVP13 board is operating at 250MHz, the XDMA IP is still configured with a 128-bit data bus to match the rest of the design. Therefore, this design cannot utilize the full bandwidth of the PCIe bus. The other inefficiency of the design is in the memory module which uses a 32-bit interface. Therefore, any data movement between the memory and the PCIe IP incurs a serialization/deserialization overhead. This gets worse with the size of the payload.

Fixing these issues involves modifying the components in the DISL component library or adding new components and updating the system specification for the design. The components can be modified independently of the rest of the modules. After the modules are updated the the configuration files that describe the components are

updated as necessary. Because the configuration files decouple the actual component implementations from the system generation, the changes to the components do not affect the system generation functionality of DISL as long as the modifications are correctly reflected in the configuration files.

The key-value store design can be optimized for better performance as follows.

- Configure the XDMA IP with the proper data bus width.

  This requires updating the TCL commands to generate the XDMA IP in the **board.tml** file for the CVP13 development board.

- If the data width of the memory module is not already parameterized, either add the parameter to the module or add a new properly parameterized memory module.

  This requires updating the `modules.tml` file.

- If the data width of the memory module is parameterized, update the parameter value in the `system.tml` file for the design.

- Changing the memory width parameter may cause the other modules that access the memory to not function properly. It might be necessary to update the parameters of other modules and add data width conversion logic to the design. These are also modifications to the `system.tml` file. The data width conversion logic could be implemented as a module in which case the `modules.tml` file has to be updated, or implemented as `'INTRINSICS'` in the `system.tml` file.

- Alternatively, the data width conversion logic can be generated. This requires adding the necessary functionality to the system generation script, `'/fpga/system_builder/build.py'`. This option may also require extending the system specification with some new notation to indicate whether a given connection requires data width conversion logic, and the data widths.

- Finally, the data width converter module placed between the PCIe subsystem and the memory module should be removed from the system specification.

The different optimizations require different levels of effort and understanding of how DISL functions. This gives the user the flexibility to choose the best option based on their expertise and the component they wish to optimize.

6.7 Summary

In this Chapter, we discussed several qualitative metrics that can be used to evaluate a hardware operating system generator. Next, we used a design example to evaluate DISL against the metrics we discussed. We have concluded that DISL allows a user to generate a system that matches the requirements of an application and port designs across devices with relatively low effort. It also allows experienced users to modify the components and the hOS generator itself with relative ease.

DISL has a few drawbacks and limitations such as the highly detailed text-based system and component descriptions making the system specifications difficult to write from scratch. But overall, the good attributes far outweigh the limitations and DISL fares well against the hOS generator evaluation metrics we have identified.

```
4 [REQUIREMENTS]
5 TOOLS = ["riscv-gnu-toolchain", "vivado"]
6 BOARDS = ["alinx_ax7a200t"]
```

```
15   [INSTANTIATIONS.refclk_ibuf]
16     MODULE = "IBUFDS_GTE2"
17
18   [INSTANTIATIONS.xdma_xc7_i]
19     MODULE = "xdma_xc7"
```

```
24   [INSTANTIATIONS.debug]
25     MODULE = "uart_axi"
26     PARAMETERS.CLOCK_FREQ_MHZ = 125
```

```
247 [INTERCONNECT]
248   STATIC = [
249         ["BOARD:sys_clk_p", "MODULE:refclk_ibuf:I"],
250         ["BOARD:sys_clk_n", "MODULE:refclk_ibuf:IB"],
251         ["MODULE:refclk_ibuf:O", "MODULE:xdma_xc7_i:sys_clk"],
```

Artix 7 (xc7a200tfbg484-2)

```
4 [REQUIREMENTS]
5 TOOLS = ["riscv-gnu-toolchain", "vivado"]
6 BOARDS = ["cvp13"]
```

```
15   [INSTANTIATIONS.refclk_ibuf]
16     MODULE = "IBUFDS_GTE4"
17     PARAMETERS.REFCLK_HROW_CK_SEL = 0
18   [INSTANTIATIONS.xdma_xcvu13p_i]
19     MODULE = "xdma_xcvu13p"
20   [INSTANTIATIONS.xdma_ext_cfg_i]
21     MODULE = "xdma_ext_cfg_space"
```

```
26   [INSTANTIATIONS.debug]
27     MODULE = "uart_axi"
28     PARAMETERS.CLOCK_FREQ_MHZ = 250
```

```
249 [INTERCONNECT]
250   STATIC = [
251         ["BOARD:sys_clk_p", "MODULE:refclk_ibuf:I"],
252         ["BOARD:sys_clk_n", "MODULE:refclk_ibuf:IB"],
253         ["MODULE:refclk_ibuf:O", "MODULE:xdma_xcvu13p_i:sys_clk_gt"],
254         ["MODULE:refclk_ibuf:ODIV2", "MODULE:xdma_xcvu13p_i:sys_clk"],
```

```
286 # Ext. cfg.
287         ["MODULE:xdma_xcvu13p_i:cfg_ext", "MODULE:xdma_ext_cfg_i:cfg_ext"],
288         ["MODULE:xdma_ext_cfg_i:clk", "CUSTOM:user_clk"],
289         ["MODULE:xdma_ext_cfg_i:rst", "CUSTOM:user_reset"],
```

Ultrascale+ (xcvu13pfigd2104-2)

Figure 6·7: Differences Between System Descriptions Targeting Two Development Boards

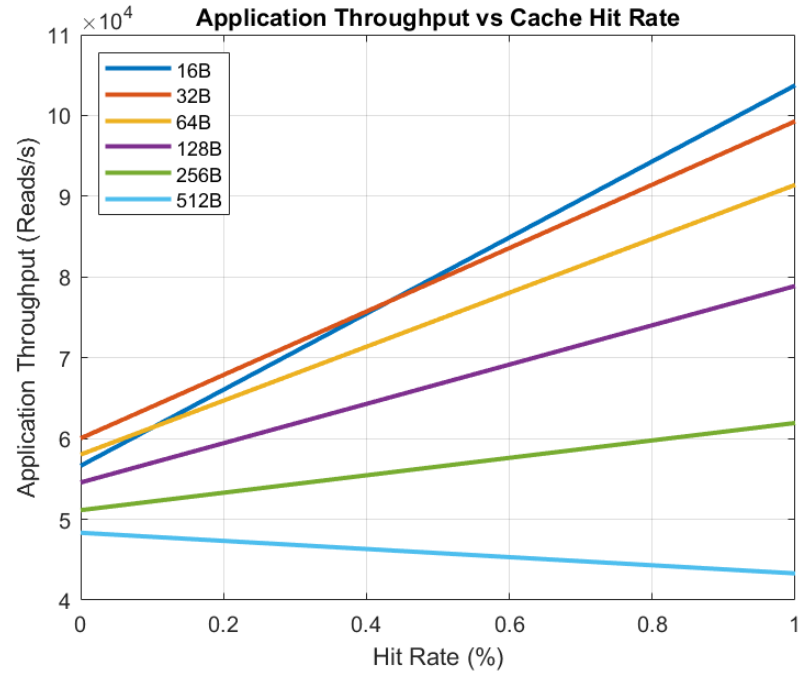


Figure 6·8: Application Throughput versus Cache Hit Rate

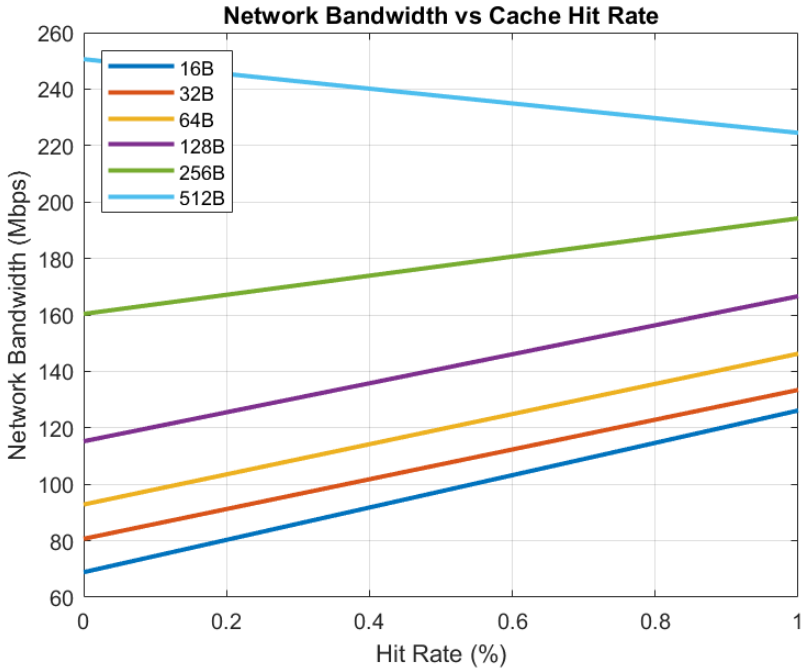


Figure 6·9: Network Bandwidth versus Cache Hit Rat

Chapter 7

Summary and Future Work

7.1 Conclusion

In this dissertation, we highlighted the importance of component design strategy to the hardware operating system generators. We began by refining the definitions of terms *hardware operating system* and *hardware operating system generator* to distinguish our work and other closely related prior work from the more software-oriented approaches. We established a set of ideal attributes for hOS generators and identified that a majority of these attributes depend on the components used by the hOS generator to build hardware operating systems. Based on these insights, we presented a set of design guidelines for hardware OS components used by system generators.

As our second contribution, we presented a PCIe subsystem design for a system generator. Our design addresses two major limitations with existing host-FPGA communication solutions; the lack of portability and the high maintenance overhead of device drivers. We have demonstrated that it is possible to use Virtio drivers that are part of standard Linux distributions as generic device drivers for FPGAs. Our results indicate that these repurposed device drivers can perform at a comparable or higher level than the vendor-provided device drivers for FPGAs. Based on these we can conclude that the legacy device drivers for FPGAs can be replaced with generic device drivers without a negative impact on performance.

Finally, we have attempted to address the open question, how to evaluate a hardware operating system generator? We recommended several qualitative metrics re-

lated to the previously identified ideal attributes for a system generator that can be used to identify a good system generator design. Through this discussion, we also highlighted how most of these metrics are either directly or indirectly related to the component design strategy. Therefore, we conclude that a good component design strategy that translates the overall goals of system generators to the hOS components is paramount for designing good system generators and hardware operating systems for FPGAs.

7.2 Future Research Directions

In this section, we discuss several future research directions related to DISL and the PCIe subsystem design presented in this dissertation.

7.2.1 Improvements to DISL

DISL is the overarching project that includes the work presented in this dissertation. We have used DISL for all the implementations in this dissertation and we also evaluated DISL based on our metrics for a good hOS generator design. Based on our insights, we present future research directions that can enhance the capabilities of DISL.

Using Another Abstraction Layer to Generate System and Component Descriptions

The main drawback of DISL we have identified is the complexity of the text-based component and system descriptions. The amount of detail to be provided in the system descriptions means that users almost always have to rely on example designs as the starting point instead of creating system descriptions from scratch. Also, the current system description method means the user is responsible for converting the application requirements to system components and the connectivity among them.

Ideally, we would like a user to be able to specify application requirements at a more abstract level and the system generator to map those to system configurations.

Therefore, an important future research direction is developing another abstraction layer capable of converting high-level descriptions of application requirements to DISL configuration files. For example, the new abstraction layer should allow a user to specify that the application requires a certain amount of memory instead of specifying BRAM or DRAM controller instances and the connectivity between those and the user logic.

There are several forms this abstraction layer could potentially take. A graphical user interface (GUI) through which the user can select the capabilities required by the application is one possibility. Another possibility is a generative AI-based tool that can generate the system descriptions based on prompts from the user.

Automating the Component Attribute Checking and Partitioning

In this dissertation, we presented a set of design guidelines to ensure the hOS components possess a set of ideal attributes. Future research can automate verifying if a given component has the desired attributes before adding it to DISL. DISL uses text-based descriptions for components, devices, and systems. The syntax for component description can be extended to facilitate automated attribute verification when adding a new component to DISL.

The current use model for DISL requires a user to manually partition a component into `Board` and `Generic` partitions. This can be a time-consuming process and also can create significant differences between components in terms of how they are partitioned. An important future research direction is automating the component partitioning.

7.2.2 Applications Enabled by the PCIe Component Design

In this section, we discuss research directions enabled by the PCIe subsystem design presented in this dissertation.

SmartNIC-based Applications

FPGAs are used as off-the-shelf platforms to build smartNICs. The PCIe subsystem we have presented implements a Virtio interface on the FPGA. By implementing a Virtio network device interface, the FPGA can be presented as a regular network device to the host OS. This allows the applications on the host to use the host operating system's network stack to send and receive packets to/from the FPGA smartNIC.

FPGA smartNIC designs implemented with vendor-provided FPGA device drivers suffer from the shortcomings of those device drivers. Because the FPGA device drivers are usually not implemented as network devices, the smartNIC cannot utilize the host's network stack. The designer can (i) implement a new device driver with the proper semantics, (ii) implement the network stack's functionality in the FPGA after moving the data to the FPGA, or (iii) lift the device semantics to the application layer. All these choices are either inefficient or take extra effort to implement.

Because our PCIe design allows using Virtio drivers and helps overcome these limitations, it enables more efficient implementations of smartNIC-based applications.

Extending the PCIe Subsystem Design with Multiple PCIe Functions

The final future research direction we discuss is extending the PCIe subsystem to support multiple physical/virtual PCIe functions. Each PCIe function could implement a different Virtio device. For instance, the same FPGA can implement a smartNIC application and a cryptographic accelerator. The applications executing on the host server can access these through two PCIe functions, first represented as

a Virtio network device and the second as a Virtio Crypto device. Figure 6·4 depicts how the multiple PCIe functions can be assigned to different virtual machines (VM) enabling direct communication between the applications executing inside the VMs and the hardware kernels on the FPGA.

Appendix A

System Definition for the Key-value Store Design

Listing A.1 shows the full system specification for the key-value store design.

```
[DESCRIPTION]
NAME = "virtio_net_device_kvs"

[REQUIREMENTS]
TOOLS = ["riscv-gnu-toolchain", "vivado"]
BOARDS = ["alinx_ax7a200t"]

[EXTERNAL_IO]
PORTS = ["sys_clk_p", "sys_clk_n", "sys_rst_n", "uart_tx","uart_rx", "
    pci_exp_txp", "pci_exp_txn", "pci_exp_rxp", "pci_exp_rxn"]

[INSTANTIATIONS]
  [INSTANTIATIONS.sys_reset_n_ibuf]
    MODULE = "IBUF"
  [INSTANTIATIONS.refclk_ibuf]
    MODULE = "IBUFDS_GTE2"

  [INSTANTIATIONS.xdma_xc7_i]
    MODULE = "xdma_xc7"
  [INSTANTIATIONS.pcie_controller_inst]
    MODULE = "pcie_controller"
    PARAMETERS.NUM_QUEUES = 2
    PARAMETERS.SLEEP_TIMER_VAL = 100
  [INSTANTIATIONS.debug]
    MODULE = "uart_axi"
    PARAMETERS.CLOCK_FREQ_MHZ = 125
    PARAMETERS.UART_BAUD_RATE_BPS = 1000000
    PARAMETERS.DATA_WIDTH = 32
  [INSTANTIATIONS.bram_intf0_access_ctrl]
    MODULE = "ss_access_controller"
    PARAMETERS.NUM_PORTS = 5
    PARAMETERS.PORT_IDX_BITS = 3
  [INSTANTIATIONS.endpoint_select_inst]
    MODULE = "endpoint_selector"
    PARAMETERS.AXI_BUS_WIDTH = 128
    PARAMETERS.AXI_ADDR_WIDTH = 64
```

```
[INSTANTIATIONS.dw_converter]
  MODULE = "data_width_converter"
  PARAMETERS.DATA_A_WIDTH = 128
  PARAMETERS.DATA_B_WIDTH = 32
  PARAMETERS.ADDR_WIDTH = 32
[INSTANTIATIONS.main_memory]
  MODULE = "bram_axi"
  PARAMETERS.ADDR_WIDTH = 14
  PARAMETERS.DATA_WIDTH = 32
  PARAMETERS.INITIALIZE = 1
  PARAMETERS.INIT_FILE  = "\"./kvs_cpu_firmware.hex\""
[INSTANTIATIONS.intf_splitter2]
  MODULE = "interface_splitter_2"
  PARAMETERS.DATA_WIDTH = 32
  PARAMETERS.ADDR_WIDTH = 32
  PARAMETERS.STRB_WIDTH = 4
[INSTANTIATIONS.intf_concat2]
  MODULE = "interface_concat_2"
  PARAMETERS.DATA_WIDTH = 32
  PARAMETERS.ADDR_WIDTH = 32
  PARAMETERS.STRB_WIDTH = 4
[INSTANTIATIONS.kvs_inst]
  MODULE = "kvs"
  PARAMETERS.LISTEN_PORT = 44000
  PARAMETERS.HASH_WIDTH = 16
  PARAMETERS.KEY_WIDTH = 32
  PARAMETERS.VALUE_WIDTH = 128
  PARAMETERS.VALUE_SIZE = 16
[INSTANTIATIONS.cpu]
  MODULE = "picorv32_axi"
  ARCH = "rv32i"
  ABI = "ilp32"
  CROSS = "riscv32-unknown-elf-"
  CROSSCFLAGS = "-O3 -Wno-int-conversion -ffreestanding -nostdlib"
  CROSSLDFLAGS = "-ffreestanding -nostdlib  -Wl,-M"
  LINKER_REQUIREMENTS = ["muldi3.S", "div.S", "riscv-asm.h"]
  MEMORY = "main_memory"
  PARAMETERS.ENABLE_INTERRUPTS = 0
  PARAMETERS.INSTRUCTION_MEMORY_STARTING_ADDRESS = 0
  PARAMETERS.INTERRUPT_HANDLER_STARTING_ADDRESS = 16
  PARAMETERS.INSTRUCTION_AND_DATA_MEMORY_SIZE_BYTES = 16384
  [INSTANTIATIONS.cpu.MAP]
    [INSTANTIATIONS.cpu.MAP.main_memory]
      ORIGIN = "0x00000000"
      LENGTH = "0x00004000"
    [INSTANTIATIONS.cpu.MAP.pci_cfg0]
      ORIGIN = "0x10000000"
      LENGTH = "0x10000000"
    [INSTANTIATIONS.cpu.MAP.pci_cfg1]
      ORIGIN = "0x20000000"
      LENGTH = "0x10000000"
    [INSTANTIATIONS.cpu.MAP.kvs_cfg]
      ORIGIN = "0x70000000"
```

```
89          LENGTH = "0x10000000"
90        [INSTANTIATIONS.cpu.MAP.debug]
91          ORIGIN = "0x80000000"
92          LENGTH = "0x10000000"
93
94  [INTRINSICS]
95    [[INTRINSICS.COMBINATIONAL]]
96      CUSTOM_SIGNAL_WIDTH = 1
97      CUSTOM_SIGNAL_NAME = "CUSTOM:user_reset"
98      INPUT_SIGNAL_1 =  "MODULE:xdma_xc7_i:axi_aresetn"
99      OPERATION = "^"
100     INPUT_SIGNAL_2 =  "1'b1"
101   [[INTRINSICS.ASSIGNMENT]]
102     CUSTOM_SIGNAL_WIDTH = 1
103     CUSTOM_SIGNAL_NAME = "CUSTOM:user_resetn"
104     INPUT_SIGNAL =  "MODULE:xdma_xc7_i:axi_aresetn"
105     SIGNAL_BITS = ""
106   [[INTRINSICS.ASSIGNMENT]]
107     CUSTOM_SIGNAL_WIDTH = 1
108     CUSTOM_SIGNAL_NAME = "CUSTOM:user_clk"
109     INPUT_SIGNAL =  "MODULE:xdma_xc7_i:axi_aclk"
110     SIGNAL_BITS = ""
111   [[INTRINSICS.CONCAT4]]
112     CUSTOM_SIGNAL_WIDTH = 5
113     CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_rvalid"
114     SIGNAL_1 = "MODULE:kvs_inst:m_axil:rvalid"
115     SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_rvalid"
116     SIGNAL_3 = "MODULE:dw_converter:b:rvalid"
117     SIGNAL_4 = "MODULE:cpu:mem:axi_rvalid"
118   [[INTRINSICS.CONCAT4]]
119     CUSTOM_SIGNAL_WIDTH = 5
120     CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_arvalid"
121     SIGNAL_1 = "MODULE:kvs_inst:m_axil:arvalid"
122     SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_arvalid"
123     SIGNAL_3 = "MODULE:dw_converter:b:arvalid"
124     SIGNAL_4 = "MODULE:cpu:mem:axi_arvalid"
125   [[INTRINSICS.CONCAT4]]
126     CUSTOM_SIGNAL_WIDTH = 5
127     CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_arready"
128     SIGNAL_1 = "MODULE:kvs_inst:m_axil:arready"
129     SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_arready"
130     SIGNAL_3 = "MODULE:dw_converter:b:arready"
131     SIGNAL_4 = "MODULE:cpu:mem:axi_arready"
132   [[INTRINSICS.CONCAT4]]
133     CUSTOM_SIGNAL_WIDTH = 5
134     CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_rready"
135     SIGNAL_1 = "MODULE:kvs_inst:m_axil:rready"
136     SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_rready"
137     SIGNAL_3 = "MODULE:dw_converter:b:rready"
138     SIGNAL_4 = "MODULE:cpu:mem:axi_rready"
139   [[INTRINSICS.CONCAT4]]
140     CUSTOM_SIGNAL_WIDTH = 5
141     CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_awvalid"
```

```
142       SIGNAL_1 = "MODULE:kvs_inst:m_axil:awvalid"
143       SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_awvalid"
144       SIGNAL_3 = "MODULE:dw_converter:b:awvalid"
145       SIGNAL_4 = "MODULE:cpu:mem:axi_awvalid"
146     [[INTRINSICS.CONCAT4]]
147       CUSTOM_SIGNAL_WIDTH = 5
148       CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_awready"
149       SIGNAL_1 = "MODULE:kvs_inst:m_axil:awready"
150       SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_awready"
151       SIGNAL_3 = "MODULE:dw_converter:b:awready"
152       SIGNAL_4 = "MODULE:cpu:mem:axi_awready"
153     [[INTRINSICS.CONCAT4]]
154       CUSTOM_SIGNAL_WIDTH = 5
155       CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_bvalid"
156       SIGNAL_1 = "MODULE:kvs_inst:m_axil:bvalid"
157       SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_bvalid"
158       SIGNAL_3 = "MODULE:dw_converter:b:bvalid"
159       SIGNAL_4 = "MODULE:cpu:mem:b_valid"
160     [[INTRINSICS.CONCAT4]]
161       CUSTOM_SIGNAL_WIDTH = 5
162       CUSTOM_SIGNAL_NAME = "CUSTOM:bram_intf0_bready"
163       SIGNAL_1 = "MODULE:kvs_inst:m_axil:bready"
164       SIGNAL_2 = "MODULE:pcie_controller_inst:m_axil_bready"
165       SIGNAL_3 = "MODULE:dw_converter:b:bready"
166       SIGNAL_4 = "MODULE:cpu:mem:b_ready"
167     [[INTRINSICS.SEQUENTIAL_HOLD]]
168       CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
169       CUSTOM_SIGNAL_NAME = "CUSTOM:cpu_axi_araddr"
170       INTERNAL_SIGNAL_NAME = "INTERNAL:CUSTOM:cpu_axi_araddr"
171       CUSTOM_SIGNAL_DEFAULT_VALUE = 0
172       CLOCK = "MODULE:cpu:clk"
173       TRIGGER = "MODULE:cpu:mem:axi_arvalid"
174       HOLD_VALUE = "MODULE:cpu:mem:axi_araddr"
175     [[INTRINSICS.SEQUENTIAL_HOLD]]
176       CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
177       CUSTOM_SIGNAL_NAME = "CUSTOM:cpu_axi_awaddr"
178       INTERNAL_SIGNAL_NAME = "INTERNAL:CUSTOM:cpu_axi_awaddr"
179       CUSTOM_SIGNAL_DEFAULT_VALUE =  0
180       CLOCK = "MODULE:cpu:clk"
181       TRIGGER = "MODULE:cpu:mem:axi_awvalid"
182       HOLD_VALUE = "MODULE:cpu:mem:axi_awaddr"
183     [[INTRINSICS.COMBINATIONAL]]
184       CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
185       CUSTOM_SIGNAL_NAME = "CUSTOM:main_mem_araddr"
186       INPUT_SIGNAL_1 =  "MODULE:main_memory:mem:axi_araddr"
187       OPERATION = ">>"
188       INPUT_SIGNAL_2 =  "2"
189     [[INTRINSICS.COMBINATIONAL]]
190       CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
191       CUSTOM_SIGNAL_NAME = "CUSTOM:main_mem_awaddr"
192       INPUT_SIGNAL_1 =  "MODULE:main_memory:mem:axi_awaddr"
193       OPERATION = ">>"
194       INPUT_SIGNAL_2 =  "2"
```

```
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:intf_concat_a0_araddr"
      INPUT_SIGNAL_1 =  "MODULE:intf_concat2:a0:araddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 =  "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg0.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:intf_concat_a0_awaddr"
      INPUT_SIGNAL_1 =  "MODULE:intf_concat2:a0:awaddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg0.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:intf_concat_a1_araddr"
      INPUT_SIGNAL_1 =  "MODULE:intf_concat2:a1:araddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg1.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:intf_concat_a1_awaddr"
      INPUT_SIGNAL_1 =  "MODULE:intf_concat2:a1:awaddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg1.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:debug_araddr"
      INPUT_SIGNAL_1 =  "MODULE:debug:a:axi_araddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.debug.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:debug_awaddr"
      INPUT_SIGNAL_1 =  "MODULE:debug:a:axi_awaddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.debug.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:kvs_araddr"
      INPUT_SIGNAL_1 =  "MODULE:kvs_inst:s_axil:araddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.kvs_cfg.ORIGIN"
    [[INTRINSICS.COMBINATIONAL]]
      CUSTOM_SIGNAL_WIDTH = "PARAMETER:cpu:ADDR_WIDTH"
      CUSTOM_SIGNAL_NAME = "CUSTOM:kvs_awaddr"
      INPUT_SIGNAL_1 =  "MODULE:kvs_inst:s_axil:awaddr"
      OPERATION = "-"
      INPUT_SIGNAL_2 = "SYSTEM:INSTANTIATIONS.cpu.MAP.kvs_cfg.ORIGIN"

  [INTERCONNECT]
    STATIC = [
      ["BOARD:sys_clk_p", "MODULE:refclk_ibuf:I"],
      ["BOARD:sys_clk_n", "MODULE:refclk_ibuf:IB"],
```

```
248     ["BOARD:uart_rx", "MODULE:debug:urx"],
249     ["BOARD:uart_tx", "MODULE:debug:utx"],
250     ["MODULE:refclk_ibuf:O", "MODULE:xdma_xc7_i:sys_clk"],
251     ["BOARD:sys_rst_n", "MODULE:sys_reset_n_ibuf:I"],
252     ["MODULE:sys_reset_n_ibuf:O", "MODULE:xdma_xc7_i:sys_rst_n"],
253     ["MODULE:xdma_xc7_i:pci_exp_txp", "BOARD:pci_exp_txp"],
254     ["MODULE:xdma_xc7_i:pci_exp_txn", "BOARD:pci_exp_txn"],
255     ["BOARD:pci_exp_rxp", "MODULE:xdma_xc7_i:pci_exp_rxp"],
256     ["BOARD:pci_exp_rxn", "MODULE:xdma_xc7_i:pci_exp_rxn"],
257     ["CUSTOM:user_clk", "MODULE:cpu:clk"],
258     ["CUSTOM:user_clk", "MODULE:debug:clk"],
259     ["CUSTOM:user_clk", "MODULE:pcie_controller_inst:clk"],
260     ["CUSTOM:user_reset", "MODULE:pcie_controller_inst:reset"],
261     ["CUSTOM:user_reset", "MODULE:debug:rst"],
262     ["CUSTOM:user_resetn", "MODULE:cpu:resetn"],
263     ["MODULE:xdma_xc7_i:m_axil", "MODULE:pcie_controller_inst:s_axil"],
264     ["MODULE:pcie_controller_inst:c2h_dsc_byp", "MODULE:xdma_xc7_i:c2h_dsc_byp
        "],
265     ["MODULE:pcie_controller_inst:h2c_dsc_byp", "MODULE:xdma_xc7_i:h2c_dsc_byp
        "],
266     ["CUSTOM:bram_intf0_rvalid", "MODULE:bram_intf0_access_ctrl:rvalid"],
267     ["CUSTOM:bram_intf0_rready", "MODULE:bram_intf0_access_ctrl:rready"],
268     ["CUSTOM:bram_intf0_arvalid", "MODULE:bram_intf0_access_ctrl:arvalid"],
269     ["CUSTOM:bram_intf0_arready", "MODULE:bram_intf0_access_ctrl:arready"],
270     ["CUSTOM:bram_intf0_awvalid", "MODULE:bram_intf0_access_ctrl:awvalid"],
271     ["CUSTOM:bram_intf0_awready", "MODULE:bram_intf0_access_ctrl:awready"],
272     ["CUSTOM:bram_intf0_bvalid", "MODULE:bram_intf0_access_ctrl:bvalid"],
273     ["CUSTOM:bram_intf0_bready", "MODULE:bram_intf0_access_ctrl:bready"],
274     ["MODULE:xdma_xc7_i:m_axi", "MODULE:endpoint_select_inst:pci_axi"],
275     ["MODULE:pcie_controller_inst:dma_engine_config_axil", "MODULE:xdma_xc7_i:
        s_axil"],
276     ["MODULE:xdma_xc7_i:c2h_sts_0", "MODULE:pcie_controller_inst:c2h_sts_0"],
277     ["MODULE:xdma_xc7_i:h2c_sts_0", "MODULE:pcie_controller_inst:h2c_sts_0"],
278     ["MODULE:pcie_controller_inst:usr_irq_req", "MODULE:xdma_xc7_i:usr_irq_req
        "],
279     ["MODULE:xdma_xc7_i:usr_irq_ack", "MODULE:pcie_controller_inst:usr_irq_ack
        "],
280 # endpoint_selector
281     ["MODULE:pcie_controller_inst:endpoint_ctrl", "MODULE:endpoint_select_inst:
        endpoint_ctrl"],
282     ["MODULE:endpoint_select_inst:vc_axi", "MODULE:pcie_controller_inst:s_axi"],
283     ["MODULE:endpoint_select_inst:ss", "MODULE:dw_converter:a"],
284     ["MODULE:endpoint_select_inst:ss:wvalid", "MODULE:pcie_controller_inst:
        s_axi_wvalid_to_mem"],
285     ["MODULE:endpoint_select_inst:ss:wstrb", "MODULE:pcie_controller_inst:
        s_axi_wstrb_to_mem"],
286     ["MODULE:dw_converter:a:rvalid", "MODULE:pcie_controller_inst:
        s_axi_rvalid_from_mem"],
287     ["MODULE:dw_converter:a:wready", "MODULE:pcie_controller_inst:
        s_axi_wready_from_mem"],
288     ["MODULE:endpoint_select_inst:ss:rready", "MODULE:pcie_controller_inst:
        s_axi_rready_to_mem"],
289     ["MODULE:endpoint_select_inst:ss:bready", "MODULE:pcie_controller_inst:
```

```
        s_axi_bready_to_mem"],
290     ["MODULE:dw_converter:a:bvalid", "MODULE:pcie_controller_inst:
        s_axi_bvalid_from_mem"],
291 # data_width_converter
292     ["CUSTOM:user_clk", "MODULE:dw_converter:clk"],
293     ["CUSTOM:user_reset", "MODULE:dw_converter:rst"],
294 # pcie_controller to interface splitter
295     ["MODULE:pcie_controller_inst:m_axil_araddr", "MODULE:intf_splitter2:
        a_araddr"],
296     ["MODULE:pcie_controller_inst:m_axil_arvalid", "MODULE:intf_splitter2:
        a_arvalid"],
297     ["MODULE:pcie_controller_inst:m_axil_arready", "MODULE:intf_splitter2:
        a_arready"],
298     ["MODULE:pcie_controller_inst:m_axil_awaddr", "MODULE:intf_splitter2:
        a_awaddr"],
299     ["MODULE:pcie_controller_inst:m_axil_awvalid", "MODULE:intf_splitter2:
        a_awvalid"],
300     ["MODULE:pcie_controller_inst:m_axil_awready", "MODULE:intf_splitter2:
        a_awready"],
301     ["MODULE:pcie_controller_inst:m_axil_wdata", "MODULE:intf_splitter2:a_wdata
        "],
302     ["MODULE:pcie_controller_inst:m_axil_wvalid", "MODULE:intf_splitter2:
        a_wvalid"],
303     ["MODULE:pcie_controller_inst:m_axil_wready", "MODULE:intf_splitter2:
        a_wready"],
304     ["MODULE:pcie_controller_inst:m_axil_wstrb", "MODULE:intf_splitter2:a_wstrb
        "],
305     ["MODULE:pcie_controller_inst:m_axil_rdata", "MODULE:intf_splitter2:a_rdata
        "],
306     ["MODULE:pcie_controller_inst:m_axil_rvalid", "MODULE:intf_splitter2:
        a_rvalid"],
307     ["MODULE:pcie_controller_inst:m_axil_rready", "MODULE:intf_splitter2:
        a_rready"],
308     ["MODULE:pcie_controller_inst:m_axil_rresp", "MODULE:intf_splitter2:a_rresp
        "],
309     ["MODULE:pcie_controller_inst:m_axil_bvalid", "MODULE:intf_splitter2:
        a_bvalid"],
310     ["MODULE:pcie_controller_inst:m_axil_bready", "MODULE:intf_splitter2:
        a_bready"],
311     ["MODULE:pcie_controller_inst:m_axil_bresp", "MODULE:intf_splitter2:a_bresp
        "],
312     ["MODULE:intf_concat2:b_awvalid", "MODULE:pcie_controller_inst:
        cfg_intf_axil_awvalid"],
313     ["MODULE:intf_concat2:b_awready", "MODULE:pcie_controller_inst:
        cfg_intf_axil_awready"],
314     ["MODULE:intf_concat2:b_awaddr", "MODULE:pcie_controller_inst:
        cfg_intf_axil_awaddr"],
315     ["MODULE:intf_concat2:b_wvalid", "MODULE:pcie_controller_inst:
        cfg_intf_axil_wvalid"],
316     ["MODULE:intf_concat2:b_wready", "MODULE:pcie_controller_inst:
        cfg_intf_axil_wready"],
317     ["MODULE:intf_concat2:b_wdata", "MODULE:pcie_controller_inst:
        cfg_intf_axil_wdata"],
```

```
318       ["MODULE:intf_concat2:b_wstrb", "MODULE:pcie_controller_inst:
          cfg_intf_axil_wstrb"],
319       ["MODULE:intf_concat2:b_bvalid", "MODULE:pcie_controller_inst:
          cfg_intf_axil_bvalid"],
320       ["MODULE:intf_concat2:b_bready", "MODULE:pcie_controller_inst:
          cfg_intf_axil_bready"],
321       ["MODULE:intf_concat2:b_bresp", "MODULE:pcie_controller_inst:
          cfg_intf_axil_bresp"],
322       ["MODULE:intf_concat2:b_arvalid", "MODULE:pcie_controller_inst:
          cfg_intf_axil_arvalid"],
323       ["MODULE:intf_concat2:b_arready", "MODULE:pcie_controller_inst:
          cfg_intf_axil_arready"],
324       ["MODULE:intf_concat2:b_araddr", "MODULE:pcie_controller_inst:
          cfg_intf_axil_araddr"],
325       ["MODULE:intf_concat2:b_rvalid", "MODULE:pcie_controller_inst:
          cfg_intf_axil_rvalid"],
326       ["MODULE:intf_concat2:b_rready", "MODULE:pcie_controller_inst:
          cfg_intf_axil_rready"],
327       ["MODULE:intf_concat2:b_rdata", "MODULE:pcie_controller_inst:
          cfg_intf_axil_rdata"],
328       ["MODULE:intf_concat2:b_rresp", "MODULE:pcie_controller_inst:
          cfg_intf_axil_rresp"],
329       ["CUSTOM:user_clk", "MODULE:bram_intf0_access_ctrl:clk"],
330       ["CUSTOM:user_reset", "MODULE:bram_intf0_access_ctrl:reset"],
331       ["CUSTOM:user_clk", "MODULE:main_memory:clk"],
332       ["CUSTOM:user_reset", "MODULE:main_memory:rst"],
333   # Key value store
334       ["CUSTOM:user_clk", "MODULE:kvs_inst:clk"],
335       ["CUSTOM:user_reset", "MODULE:kvs_inst:rst"]
336     ]
337
338     OVERRIDES = [ # replace port signal assignments at the end with the overrides
339       ["MODULE:refclk_ibuf:CEB","0"],
340       ["MODULE:xdma_xc7_i:m_axi_bid", "0"],
341       ["MODULE:xdma_xc7_i:m_axi_rid", "0"],
342       ["MODULE:xdma_xc7_i:s_axil_awprot", "0"],
343       ["MODULE:xdma_xc7_i:s_axil_arprot", "0"],
344       ["MODULE:dw_converter:b:rresp", "0"],
345       ["MODULE:pcie_controller_inst:interrupt_usr_ack", "0"],
346       ["MODULE:main_memory:mem:axi_araddr", "CUSTOM:main_mem_araddr"],
347       ["MODULE:main_memory:mem:axi_awaddr", "CUSTOM:main_mem_awaddr"],
348       ["MODULE:intf_concat2:a0:araddr", "CUSTOM:intf_concat_a0_araddr"],
349       ["MODULE:intf_concat2:a0:awaddr", "CUSTOM:intf_concat_a0_awaddr"],
350       ["MODULE:intf_concat2:a1:araddr", "CUSTOM:intf_concat_a1_araddr"],
351       ["MODULE:intf_concat2:a1:awaddr", "CUSTOM:intf_concat_a1_awaddr"],
352       ["MODULE:debug:a:axi_araddr", "CUSTOM:debug_araddr"],
353       ["MODULE:debug:a:axi_awaddr", "CUSTOM:debug_awaddr"],
354       ["MODULE:kvs_inst:s_axil:araddr", "CUSTOM:kvs_araddr"],
355       ["MODULE:kvs_inst:s_axil:awaddr", "CUSTOM:kvs_awaddr"],
356       ["MODULE:intf_splitter2:b0:rresp", "0"],
357       ["MODULE:intf_splitter2:b1:rresp", "0"]
358     ]
359
```

```
[INTERCONNECT.DYNAMIC."MODULE:cpu:mem"]
  GROUP_SELECT = ""
  HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
  WRITE_RESPONSE"]
  [[INTERCONNECT.DYNAMIC."MODULE:cpu:mem".GROUPS]]
    INTERCONNECT_TYPE = "ONE_TO_MANY"
    ADDRESS_MAP = [
      "SYSTEM:INSTANTIATIONS.cpu.MAP.main_memory MODULE:main_memory:mem",
      "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg0 MODULE:intf_concat2:a0",
      "SYSTEM:INSTANTIATIONS.cpu.MAP.pci_cfg1 MODULE:intf_concat2:a1",
      "SYSTEM:INSTANTIATIONS.cpu.MAP.kvs_cfg MODULE:kvs_inst:s_axil",
      "SYSTEM:INSTANTIATIONS.cpu.MAP.debug MODULE:debug:a"
    ]
    HANDSHAKE_MAP = [
      "WRITE_ADDRESS MODULE:cpu:mem:axi_awaddr",
      "READ_ADDRESS MODULE:cpu:mem:axi_araddr",
      "WRITE_DATA CUSTOM:cpu_axi_awaddr",
      "WRITE_RESPONSE CUSTOM:cpu_axi_awaddr",
      "READ_DATA CUSTOM:cpu_axi_araddr"
    ]

[INTERCONNECT.DYNAMIC."MODULE:debug:a"]
  GROUP_SELECT = ""
  HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
  WRITE_RESPONSE"]
  [[INTERCONNECT.DYNAMIC."MODULE:debug:a".GROUPS]]
    INTERCONNECT_TYPE = "ONE_TO_ONE"
    INTERFACE = "MODULE:cpu:mem"

[INTERCONNECT.DYNAMIC."MODULE:kvs_inst:s_axil"]
  GROUP_SELECT = ""
  HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
  WRITE_RESPONSE"]
  [[INTERCONNECT.DYNAMIC."MODULE:kvs_inst:s_axil".GROUPS]]
    INTERCONNECT_TYPE = "ONE_TO_ONE"
    INTERFACE = "MODULE:cpu:mem"

[INTERCONNECT.DYNAMIC."MODULE:kvs_inst:m_axil"]
  GROUP_SELECT = ""
  HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
  WRITE_RESPONSE"]
  [[INTERCONNECT.DYNAMIC."MODULE:kvs_inst:m_axil".GROUPS]]
    INTERCONNECT_TYPE = "ONE_TO_ONE"
    INTERFACE = "MODULE:main_memory:mem"

[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem"]
  GROUP_SELECT = "MODULE:bram_intf0_access_ctrl:group_select"
  HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
  WRITE_RESPONSE"]
  [[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem".GROUPS]]
    SELECT_VALUE = 0
    INTERCONNECT_TYPE = "ONE_TO_ONE"
    INTERFACE = "MODULE:cpu:mem"
```

```
408       [[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem".GROUPS]]
409         SELECT_VALUE = 1
410         INTERCONNECT_TYPE = "ONE_TO_ONE"
411         INTERFACE = "MODULE:dw_converter:b"
412       [[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem".GROUPS]]
413         SELECT_VALUE = 2
414         INTERCONNECT_TYPE = "ONE_TO_ONE"
415         INTERFACE = "MODULE:intf_splitter2:b0"
416       [[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem".GROUPS]]
417         SELECT_VALUE = 3
418         INTERCONNECT_TYPE = "ONE_TO_ONE"
419         INTERFACE = "MODULE:intf_splitter2:b1"
420       [[INTERCONNECT.DYNAMIC."MODULE:main_memory:mem".GROUPS]]
421         SELECT_VALUE = 4
422         INTERCONNECT_TYPE = "ONE_TO_ONE"
423         INTERFACE = "MODULE:kvs_inst:m_axil"
424
425     [INTERCONNECT.DYNAMIC."MODULE:dw_converter:b"]
426       GROUP_SELECT = "MODULE:bram_intf0_access_ctrl:group_select"
427       HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
          WRITE_RESPONSE"]
428       [[INTERCONNECT.DYNAMIC."MODULE:dw_converter:b".GROUPS]]
429         SELECT_VALUE = 1
430         INTERCONNECT_TYPE = "ONE_TO_ONE"
431         INTERFACE = "MODULE:main_memory:mem"
432
433     [INTERCONNECT.DYNAMIC."MODULE:intf_splitter2:b0"]
434       GROUP_SELECT = ""
435       HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
          WRITE_RESPONSE"]
436       [[INTERCONNECT.DYNAMIC."MODULE:intf_splitter2:b0".GROUPS]]
437         INTERCONNECT_TYPE = "ONE_TO_ONE"
438         INTERFACE = "MODULE:main_memory:mem"
439
440     [INTERCONNECT.DYNAMIC."MODULE:intf_splitter2:b1"]
441       GROUP_SELECT = ""
442       HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
          WRITE_RESPONSE"]
443       [[INTERCONNECT.DYNAMIC."MODULE:intf_splitter2:b1".GROUPS]]
444         INTERCONNECT_TYPE = "ONE_TO_ONE"
445         INTERFACE = "MODULE:main_memory:mem"
446
447     [INTERCONNECT.DYNAMIC."MODULE:intf_concat2:a0"]
448       GROUP_SELECT = ""
449       HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
          WRITE_RESPONSE"]
450       [[INTERCONNECT.DYNAMIC."MODULE:intf_concat2:a0".GROUPS]]
451         INTERCONNECT_TYPE = "ONE_TO_ONE"
452         INTERFACE = "MODULE:cpu:mem"
453
454     [INTERCONNECT.DYNAMIC."MODULE:intf_concat2:a1"]
455       GROUP_SELECT = ""
456       HANDSHAKES = ["WRITE_ADDRESS", "WRITE_DATA", "READ_ADDRESS", "READ_DATA", "
```

```
        WRITE_RESPONSE"]
457     [[INTERCONNECT.DYNAMIC."MODULE:intf_concat2:a1".GROUPS]]
458       INTERCONNECT_TYPE = "ONE_TO_ONE"
459       INTERFACE = "MODULE:cpu:mem"
```

Listing A.1: system.tml file Describing the Key-Value Store Design

References

Abts, D., Kimmell, G., Ling, A., Kim, J., Boyd, M., Bitar, A., Parmar, S., Ahmed, I., DiCecco, R., Han, D., et al. (2022). A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 567–580.

Agne, A., Happe, M., Keller, A., Lübbers, E., Plattner, B., Platzner, M., and Plessl, C. (2013). ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71.

Alibaba (2024). ApsaraDB for Memcache. https://www.alibabacloud.com/product/apsaradb-for-memcache. Last accessed on August 28, 2024.

Alibaba Cloud (2018). Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances.

https://alibaba-cloud.medium.com/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances-74b9aeac98ed. Last accessed on August 28, 2024.

Amazon (2024). Amazon ElastiCache. https://aws.amazon.com/elasticache/. Last accessed on August 28, 2024.

Amazon Web Services (2024). Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Last accessed on August 28, 2024.

AMD (2024a). AMD AI Engine Technology. https://www.xilinx.com/products/technology/ai-engine.html. Last accessed on July 27, 2024.

AMD (2024b). AMD Ryzen AI Processors. https://www.amd.com/en/products/processors/business-systems/ryzen-ai.html. Last accessed on July 27, 2024.

AMD (2024c). AMD XDNA Architecture. https://www.amd.com/en/technologies/xdna.html. Last accessed on July 27, 2024.

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

Anderson, E., Peck, W., Stevens, J., Agron, J., Baijot, F., Warn, S., and Andrews, D. (2007). Supporting high level language semantics within hardware resident threads. In *2007 International Conference on Field Programmable Logic and Applications*, pages 98–103. IEEE.

ARM (2023). AMBA AXI Protocol Specification. https://developer.arm.com/documentation/ihi0022/latest/. Last accessed on August 11, 2024.

ARM (2024). big.LITTLE: Balancing Power Efficiency and Performance. https://www.arm.com/technologies/big-little. Last accessed on July 27, 2024.

Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. (2012). Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64. doi: https://doi.org/10.1145/2318857.2254766.

Bandara, S., Cherry, N., and Herbordt, M. (2024a). Fully Transparent Client-Side Caching for Key-Value Store Applications Using FPGAs. In *IEEE High Performance Extreme Computing Conference*.

Bandara, S., Ducimo, A., Wu, C., and Herbordt, M. (2024b). Long-Range MD Electrostatics Force Computation on FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 35(10):1690–1707. doi: 10.1109/TPDS.2024.3434347.

Bandara, S., Sanaullah, A., Tahir, Z., Drepper, U., and Herbordt, M. (2022). Enabling VirtIO Driver Support on FPGAs. In *8th International Workshop on Heterogeneous High Performance Reconfigurable Computing*. doi: 10.1109/H2RC56700.2022.00006.

Bandara, S., Sanaullah, A., Tahir, Z., Drepper, U., and Herbordt, M. (2024c). Performance Evaluation of VirtIO Device Drivers for Host-FPGA PCIe Communication. In *31st Reconfigurable Architectures Workshop (RAW)*. doi: 10.1109/IPDPSW63119.2024.00043.

Bittware (2024). CVP-13 FPGA Cryptocurrency Mining Board. https://www.bittware.com/cvp-13/. Last accessed on August 28, 2024.

Blott, M., Karras, K., Liu, L., Vissers, K., Bär, J., and István, Z. (2013). Achieving 10gbps line-rate key-value stores with {FPGAs}. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*.

Bobda, C., Mandebi, J., Chow, P., Ewais, M., Tarafdar, N., Vega, J., Eguro, K., Koch, D., Handagala, S., Leeser, M., Herbordt, M., Shahzad, H., Hofstee, P., Ringlein, B., Szefer, J., Sanaullah, A., and Tessier, R. (2022). The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Transactions on Reconfigurable Technology and Systems*, 15(3):1–42. doi: 10.1145/3506713.

Burstein, I. (2021). Nvidia Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20.

Caulfield, A., Chung, E., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. (2016). A cloud-scale acceleration architecture. In *49th IEEE/ACM Int. Symp. Microarchitecture*, pages 1–13.

Chalamalasetti, S. R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P., and Margala, M. (2013). An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 245–254.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26.

Chiu, M. (2011). *Accelerating Molecular Dynamics Simulations with High Performance Reconfigurable Systems*. PhD thesis, Department of Electrical and Computer Engineering, Boston University.

Choi, J., Lian, R., Li, Z., Canis, A., and Anderson, J. (2018). Accelerating Memcached on AWS Cloud FPGAs. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–8.

Coyote (2024). Coyote OS for FPGAs. https://github.com/fpgasystems/Coyote. Last accessed on August 11, 2024.

Dastidar, J., Riddoch, D., Moore, J., Pope, S., and Wesselkamper, J. (2023). The AMD 400-G adaptive SmartNIC system on chip: a technology preview. *IEEE Micro*, 43(3):40–49.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220.

Dennard, R. H., Gaensslen, F. H., Yu, H.-N., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268.

Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. (2014). {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414.

Fan, B., Andersen, D. G., and Kaminsky, M. (2013). {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384.

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. (2018). Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66.

Fleming, K. and Adler, M. (2016). The LEAP FPGA operating system. *FPGAs for software programmers*, pages 245–258.

Fukuda, E. S., Inoue, H., Takenaka, T., Kim, D., Sadahisa, T., Asai, T., and Motomura, M. (2014). Caching memcached at reconfigurable network interface. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE.

Geng, T., Wang, T., Wu, C., Li, Y., Yang, C., Wu, W., Li, A., and Herbordt, M. (2021). O3BNN-R: An Out-Of-Order Architecture for High-Performance and Regularized BNN Inference. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):199–213. doi: 10.1109/TPDS.2020.3013637.

Ghigoff, Y., Sopena, J., Lazri, K., Blin, A., and Muller, G. (2021). {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501.

Gokhale, M., Frigo, J., Ahrens, C., Tripp, J., and Minnich, R. (2004). Monte Carlo radiative heat transfer simulation. In *IEEE Conference on Field Programmable Logic and Applications*, pages 95–104.

Gokhale, M. and Graham, P. (2005). *Reconfigurable Computing: Accelerating Computation with Field Programmable Gate Arrays*. Springer.

Google Cloud (2024a). Google Cloud Memorystore. https://cloud.google.com/memorystore. Last accessed on August 28, 2024.

Google Cloud (2024b). Tensor Processing Units (TPUs). https://cloud.google.com/tpu. Last accessed on July 27, 2024.

Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022a). A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *International Conference on Field-Programmable Logic and Applications*. DOI: 10.1109/FPL57034.2022.00071.

Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022b). FCsN: A FPGA-Centric SmartNIC Framework for Neural Networks. In *30th IEEE International Symposium on Field-Programmable Custom Computing Machines*. DOI: 10.1109/FCCM53951.2022.9786193.

Guo, A., Hao, Y., Wu, C., Haghi, P., Pan, Z., Si, M., Tao, D., Li, A., Herbordt, M., and Geng, T. (2023). Software-hardware co-design of heterogeneous smartnic system for recommendation models inference and training. In *ICS 2023: International Conference on Supercomputing*. DOI = 10.1145/3577193.3593724.

Haghi, P., Geng, T., Guo, A., Wang, T., and Herbordt, M. (2020). Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *IEEE Conference on Field Programmable Technology*.

Haghi, P., Guo, A., Xiong, Q., Yang, C., Geng, T., Broaddus, J., Marshall, R., Schafer, D., Skjellum, A., and Herbordt, M. (2022). Reconfigurable switches for high performance and flexible mpi collectives. *Concurrency and Computation: Practice and Experience*, 34(2). doi: 10.1002/cpe.6769.

Haghi, P., Krska, W., Tan, C., Geng, T., Chen, P., Greenwood, C., Guo, A., Hines, T., Wu, C., Li, A., Skjellum, A., and Herbordt, M. (2023). Flash: Fpga-accelerated smart switches with gcn case study. In *37th ACM International Conference on Supercomputing (ICS)*. DOI = 10.1145/3577193.3593739.

Haghi, P., Tan, C., Guo, A., Wu, C., Liu, D., Li, A., Skjellum, A., Geng, T., and Herbordt, M. (2024). Smartfuse: Reconfigurable smart switches to accelerate fused collectives in hpc applications. In *38th ACM International Conference on Supercomputing (ICS)*. DOI: 10.1145/3650200.3656616.

Hauck, S. and DeHon, A. (2008). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Morgan Kaufmann.

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

Hennessy, J. L. and Patterson, D. A. (2019). A new golden age for computer architecture innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way. https://cacm.acm.org/research/a-new-golden-age-for-computer-architecture/.

Intel (2011). Tick/Tock Predictability Continues.

https://download.intel.com/newsroom/kits/idf/2011_fall/pdfs/Kirk_Skaugen_DCSG_MegaBriefing.pdf#page=21. Last accessed on July 27, 2024.

Intel (2017). Open Programmable Acceleration Engin. https://opae.github.io/latest/index.html. Last accessed on August 28, 2024.

Intel (2023a). P-Tile Avalon Streaming Intel FPGA IP for PCI Express User Guide. 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. Last accessed on August 28, 2024.

Intel (2023b). What Is Performance Hybrid Architecture? https://www.intel.com/content/www/us/en/support/articles/000091896/processors.html. Last accessed on July 27, 2024.

István, Z., Alonso, G., and Singla, A. (2018). Providing multi-tenant services with fpgas: Case study on a key-value store. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 119–1195. IEEE.

Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N. S., Ouyang, X., Wang, H., Sur, S., et al. (2011). Memcached design on high performance rdma capable interconnects. In *2011 International Conference on Parallel Processing*, pages 743–752. IEEE.

Koeher, S., Curreri, J., and George, A. (2008). Performance analysis challenges and framework for high performance reconfigurable computing. *Parallel Computing*, 34(4-5):217–230.

Korinth, J., de la Chevallerie, D., and Koch, A. (2015). An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 195–198. IEEE.

Korolija, D., Roscoe, T., and Alonso, G. (2020). Do OS abstractions make sense on FPGAs? In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 991–1010.

Krishnan, V., Serres, O., and Blocksome, M. (2021). Configurable Network Protocol Accelerator (COPA). 41(1).

Lant, J., Navaridas, J., Lujan, M., and Goodacre, J. (2020). Toward FPGA-Based HPC: Advancing Interconnect Technology. *IEEE Micro*, 40(1):25–34.

Lavasani, M., Angepat, H., and Chiou, D. (2013). An FPGA-based In-line Accelerator for Memcached. *IEEE Computer architecture letters*, 13(2):57–60.

Li, A., Geng, T., Wang, T., Herbordt, M., Song, S., and Barker, K. (2019). BSTC: A Novel Binarized-Soft-Tensor-Core Design for Accelerating Bit-Based Approximated Neural Nets. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. doi: 10.1145/ 3295500.3356169.

Liang, W., Yin, W., Kang, P., and Wang, L. (2016). Memory Efficient and High Performance Key-value Store on FPGA Using Cuckoo Hashing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE.

Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). {MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444.

Liu, Y., Ma, J., Zhang, Z., Li, L., Qi, Z., and Guan, H. (2021). MEGATRON: Software-Managed Device TLB for Shared-Memory FPGA Virtualization. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1213–1218. IEEE.

Lübbers, E. and Platzner, M. (2009). ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):1–33.

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

Mbongue, J. M., Hategekimana, F., Kwadjo, D. T., and Bobda, C. (2018). FPGA Virtualization in Cloud-based Infrastructures over Virtio. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 242–245. doi: 10.1109/ICCD.2018.00044.

Mbongue, J. M., Kwadjo, D. T., Shuping, A., and Bobda, C. (2021). Deploying multi-tenant FPGAs within Linux-based cloud infrastructure. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(2):1–31. https://doi.org/10.1145/3474058.

Memcached (2017). Binaryprotocolrevamped. https://github.com/memcached/memcached/wiki/BinaryProtocolRevamped. Last accessed on August 28, 2024.

Memcached (2024). Memcached - A Distributed Memory Object Caching System. https://memcached.org/. Last accessed on August 28, 2024.

Microsoft (2024a). Azure Cache for Redis. https://azure.microsoft.com/en-ca/services/cache. Last accessed on August 28, 2024.

Microsoft (2024b). Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/. Last accessed on August 28, 2024.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8). Available at: http://cva.stanford.edu/classes/cs99s/papers/moore-crammingmorecomponents.pdf.

Munafo, R., Shahzad, H., Sanaullah, A., Arora, S., Drepper, U., and Herbordt, M. (2023). Improved Models for Policy-Agent Learning of Compiler Directives in HLS. In *IEEE High Performance Extreme Computing Conference*. doi: 10.1109/HPEC58863.2023.10363530.

Nimbix (2024). Cloud FPGA Accelerators: Turbocharge Your Workflows for Efficient Processing. https://www.nimbix.net/fpga-compute/. Last accessed on August 28, 2024.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. (2013). Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398.

Nvidia (2024). NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/. Last accessed on July 27, 2024.

Oliver, N., Sharma, R. R., Chang, S., Chitlur, B., Garcia, E., Grecco, J., Grier, A., Ijih, N., Liu, Y., Marolia, P., et al. (2011). A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85. IEEE.

Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., et al. (2015). The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55.

Patel, R., Haghi, P., Jain, S., Kot, A., Krishnan, V., Varia, M., and Herbordt, M. (2022). COPA Use Case: Distributed Secure Joint Computation. In *30th IEEE International Symposium on Field-Programmable Custom Computing Machines*. doi: 10.1109/FCCM53951.2022.9786156.

Patel, R., Wolfe, P.-F., Munafo, R., Varia, M., and Herbordt, M. (2020). Arithmetic and Boolean Secret Sharing MPC on FPGAs in the Data Center. In *IEEE High Performance Extreme Computing Conference*. doi: TBD.

Peck, W., Anderson, E., Agron, J., Stevens, J., Baijot, F., and Andrews, D. (2006). Hthreads: A Computational Model for Reconfigurable Devices. In *2006 International conference on field programmable logic and applications*, pages 1–4. IEEE.

Prabhakar, R., Jairath, S., and Shin, J. L. (2022). SambaNova SN10 RDU: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 350–352. IEEE.

Preston-Werner, Tom and Gedam, Pradyun et al. (2023). TOML:Tom's Obvious, Minimal Language. https://github.com/toml-lang/toml.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. (2016). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *Communications of the ACM*, 59(11):114–122.

Pérez Martín, E. (2020). Virtio devices and drivers overview: Who is who. https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone.

Red Hat (2024). Red Hat CoDes Lab - DISL. https://github.com/rh-codes-lab/DISL/tree/main.

Red Hat Customer Portal (2024). Red Hat Enterprise Linux Release Dates. https://access.redhat.com/articles/3078. Last accessed on August 28, 2024.

Redis (2024). Redis - The Real-time Data Platform. https://redis.io/. Last accessed on August 28, 2024.

PCI SIG Org. (2010). PCI Express Base Specification Revision 3.0. https://pcisig.com/specifications.

Richmond, D., Prost-Boucle, A., and O'Brien, J. (2022). RIFFA. https://github.com/KastnerRG/riffa. Last accessed on August 28, 2024.

Rodríguez, A., Valverde, J., Portilla, J., Otero, A., Riesgo, T., and De la Torre, E. (2018). FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The $ARTICo^3$ Famework. *Sensors*, 18(6):1877.

RSPwFPGAs (2020). Virtio-FPGA-Bridge: Virtio front-end and back-end bridge, implemented with FPGA. https://github.com/RSPwFPGAs/virtio-fpga-bridge. Last accessed on August 28, 2024.

Sajjadinasab, R., Arora, S., Drepper, U., Sanaullah, A., and Herbordt, M. (2024a). A Graph-Based Algorithm for Optimizing GCC Compiler Flag Settings. In *IEEE High Performance extreme Computing Conference (HPEC)*. doi: TBD.

Sajjadinasab, R., Rastaghi, H., Shahzad, H., Arora, S., Drepper, U., and Herbordt, M. (2024b). Further Optimizations and Analysis of Smith-Waterman with Vector Extensions. In *23rd IEEE International Workshop on High Performance Computational Biology (HiCOMB)*. doi: TBD.

Sakakibara, Y., Nakamura, K., and Matsutani, H. (2017). An fpga nic based hardware caching for blockchain. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, pages 1–6.

Sanka, A. I., Chowdhury, M. H., and Cheung, R. C. (2021). Efficient high-performance fpga-redis hybrid nosql caching system for blockchain scalability. *Computer Communications*, 169:81–91.

Shahzad, H., Sanaullah, A., , Arora, S., Drepper, U., and Herbordt, M. (2024a). Neural Network based GCC Cost Model for Accelerating Compiler Tuning Workloads. In *IEEE High Performance Extreme Computing Conference*.

Shahzad, H., Sanaullah, A., Arora, S., Drepper, U., and Herbordt, M. (2024b). AutoAnnotate: Reinforcement Learning based Code Annotation for High Level Synthesis. In *25th International Symposium on Quality Electronic Design*. DOI: 10.1109/ISQED60706.2024.10528738.

Shahzad, H., Sanaullah, A., Arora, S., Munafo, R., Yao, X., Drepper, U., and Herbordt, M. (2022). Reinforcement Learning Strategies for Compiler Optimization in High Level Synthesis. In *The Eighth Workshop on the LLVM Compiler Infrastructure in HPC*. DOI: 10.1109/LLVM-HPC56686.2022.00007.

Shahzad, H., Sanaullah, A., and Herbordt, M. (2021). Survey and Future Trends for FPGA Cloud Architectures. In *IEEE High Performance Extreme Computing Conference*. DOI: 10.1109/HPEC49654.2021.9622807.

Sheng, J., Humphries, B., Zhang, H., and Herbordt, M. (2015). FPGA-Centric Clusters. In *Boston Area Computer Architecture Workshop*.

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2017). Collective Communication on FPGA Clusters with Static Scheduling. *ACM SIGARCH Computer Architecture News*, 44(4). doi: 10.1145/ 3039902.3039904.

Sheng, J., Yang, C., and Herbordt, M. (2018). High Performance Dynamic Communication on Reconfigurable Clusters. In *28th International Conference on Field Programmable Logic and Applications*. doi: 10.1109/ FPL.2018.00044.

Silicom (2021). Silicom C5010X data center NIC. https://www.silicom-usa.com/wp-content/uploads/2021/12/C5010X-Data-Center-FPGA-IPU-NIC.pdf. Last accessed on August 28, 2024.

So, H. K.-H. (2007). *BORPH: An operating system for FPGA-based reconfigurable computers*. Technical Report, EECS Dept, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.pdf.

So, H. K.-H. and Brodersen, R. (2008). A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers Using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–28.

So, H. K.-H. and Brodersen, R. W. (2006). Improving usability of FPGA-based reconfigurable computers through operating system support. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6. IEEE.

Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *2008 International Conference on Field Programmable Logic and Applications*, pages 341–346. doi: 10.1109/ FPL.2008.4629955.

Sukhwani, B. and Herbordt, M. (2010). FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195. doi: 10.1049/ iet-cdt.2009.0013.

Sundar, N., Burres, B., Li, Y., Minturn, D., Johnson, B., and Jain, N. (2023). An In-depth Look at the Intel IPU E2000. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 162–164. IEEE.

Tahir, Z., Sanaullah, A., Bandara, S., Drepper, U., and Herbordt, M. (2024a). Multi-core Multi-rule VeBPF Firewall for Secure FPGA IoT Device Deployments. In *31st Reconfigurable Architectures Workshop (RAW)*. doi: 10.1109/IPDPSW63119.2024.00053.

Tahir, Z., Sanaullah, A., Bandara, S., Drepper, U., and Herbordt, M. (2024b). Multi-core Multi-rule VeBPF Firewall for Secure FPGA IoT Device Deployments. In *IEEE High Performance Extreme Computing Conference*.

Tang, W., Lu, Y., Xiao, N., Liu, F., and Chen, Z. (2017). Accelerating redis with rdma over infiniband. In *Data Mining and Big Data: Second International Conference, DMBD 2017, Fukuoka, Japan, July 27–August 1, 2017, Proceedings 2*, pages 472–483. Springer.

The Linux Kernel Archives (2024). Active Kernel Releases. https://www.kernel.org/category/releases.html. Last accessed on August 28, 2024.

Tsirkin, M. S. and Huck, C. (2022). Virtual I/O device (VIRTIO) version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html.

Vaishnav, A., Pham, K. D., Powell, J., and Koch, D. (2020). FOS: A modular FPGA operating system for dynamic workloads. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(4):1–28.

VanCourt, T., Gu, Y., and Herbordt, M. (2004). FPGA acceleration of rigid molecule interactions. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 300–301. doi: 10.1109/ FCCM.2004.33.

VanCourt, T. and Herbordt, M. (2006). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10. doi: 10.1155/ ASP/2006/97950.

Vogel, P., Marongiu, A., and Benini, L. (2018). Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. *IEEE Transactions on Computers*, 68(4):510–525.

Wenzel, J. and Hochberger, C. (2016). RapidSoC: Short Turnaround Creation of FPGA Based SoCs. In *Proceedings of the 27th international symposium on rapid system prototyping: shortening the path from specification to prototype*, pages 86–92.

Wikipedia (2024a). Dennard Scaling. https://en.wikipedia.org/wiki/Dennard_scaling. Last accessed on July 27, 2024.

Wikipedia (2024b). Moore's law. https://en.wikipedia.org/wiki/Moore's_law. Last accessed on July 27, 2024.

Wikipedia (2024c). Tick-tock model. https://en.wikipedia.org/wiki/Tick-tock_model. Last accessed on July 27, 2024.

Wolfe, P.-F., Patel, R., Munafo, R., Varia, M., and Herbordt, M. (2020). Secret Sharing MPC on FPGAs in the Datacenter. In *IEEE Conference on Field Programmable Logic and Applications*.

Wu, C., Bandara, S., Geng, T., Guo, A., Haghi, P., Sherman, W., Sachdeva, V., and Herbordt, M. (2022). Optimized Mappings for Symmetric Range-Limited Molecular Force Calculations on FPGAs. In *International Conference on Field-Programmable Logic and Applications*. DOI: 10.1109/FPL57034.2022.00026.

Wu, C., Geng, T., Bandara, S., Yang, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2021). Upgrade of FPGA Range-Limited Molecular Dynamics to Handle Hundreds of Processors. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. doi: 10.1109/FCCM51124.2021.00024.

Wu, C., Geng, T., Guo, A., Bandara, S., Haghi, P., Liu, C., Li, A., and Herbordt, M. (2023). FASDA: An FPGA-Aided, Scalable and Distributed Accelerator for Range-Limited Molecular Dynamics. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. DOI: 10.1145/3581784.3607100.

Wu, C., Yang, C., Bandara, S., Geng, T., Haghi, P., Li, A., and Herbordt, M. (2024). FPGA-Accelerated Range-Limited Molecular Dynamics. *IEEE Transactions on Computers*, 73(6):1544–1558. doi: 10.1109/TC.2024.3375613.

Xilinx (2020). 7 Series FPGAs Integrated Block for PCI Express v3.3. https://docs.xilinx.com/v/u/en-US/pg054-7series-pcie.

Xilinx (2022a). Artix-7 FPGA family. https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

Xilinx (2022b). DMA/Bridge Subsystem for PCI Express v4.1. https://docs.xilinx.com/r/en-US/pg195-pcie-dma.

Xilinx (2023). AMD OpenNIC Shell. https://github.com/Xilinx/open-nic-shell. Last accessed on July 28, 2024.

Xilinx (2024). Xilinx DMA IP Reference drivers. https://github.com/Xilinx/dma_ip_drivers. Last accessed on August 28, 2024.

Xilinx Inc. (2022). Xilinx Run Time for FPGA. https://github.com/Xilinx/XRT.

XILLYBUS (2022). Xillybus IP core product brief. https://xillybus.com/downloads/xillybus_product_brief.pdf. Last accessed on August 28, 2024.

Xiong, Q., Yang, C., Patel, R., Geng, T., Skjellum, A., and Herbordt, M. (2019). GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266. doi: 10.1109/FCCM.2019.00042.

Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. (2013). Characterizing facebook's memcached workload. *IEEE Internet Computing*, 18(2):41–49.

Zhang, J., Xiong, Y., Xu, N., Shu, R., Li, B., Cheng, P., Chen, G., and Moscibroda, T. (2017). The Feniks FPGA operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7.

Zink, M., Irwin, D., Cecchet, E., Saplakoglu, H., Krieger, O., Herbordt, M., Daitzman, M., Desnoyers, P., Leeser, M., and Handagala, S. (2021). The Open Cloud Testbed (OCT): A Platform for Research into new Cloud Technologies. In *IEEE International Conference on Cloud Networking (IEEE CloudNet)*. doi: 10.1109/CloudNet53349.2021.9657109.

CURRICULUM VITAE