DISS. ETH NO. 29953

Abstractions for Modern Heterogeneous Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

DARIO KOROLIJA

MSc in Electrical Engineering
EPFL

born on 18.04.1993

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zurich)
Prof. Dr. Timothy Roscoe (ETH Zurich)
Prof. Dr. Lana Josipović (ETH Zurich)
Dr. Michaela Blott (AMD)

2024

IKS-Lab
Institute for Computing Platforms
ETH Department of Computer Science

Dario Korolija

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 29953

examiner:
Prof. Dr. Gustavo Alonso
co-examiners:
Prof. Dr. Timothy Roscoe
Prof. Dr. Lana Josipović
Dr. Michaela Blott

Examination date: December 19, 2023

Abstractions for Modern Heterogeneous Systems

Abstract

The hardware landscape is undergoing a significant transformation, primarily fueled by the ever-increasing demands for data processing. Modern computing systems are increasingly incorporating a wide range of heterogeneous processing units, signifying a departure from the traditional Von-Neumann architectures. A prime illustration of this diversification is the widespread adoption of hybrid computing platforms that combine a general-purpose CPU with a Field-Programmable Gate Array (FPGA), now becoming a standard feature in modern data centers and cloud environments.

These platforms significantly improve performance and energy efficiency across diverse workloads, thanks to their notable flexibility in accommodating a wide range of processing demands. However, their heterogeneity presents considerable challenges in terms of software compatibility, programming, and overall system design. Navigating these complex platforms to fully exploit the potential of the diverse hardware can be challenging. Notably, system software for these devices lags behind, lacking the traditional abstractions commonly found in conventional operating systems.

This thesis bridges this gap between modern heterogeneous hardware and its corresponding system software. To this end, we introduce Coyote, a comprehensive FPGA "shell" that provides a core set of essential features and abstractions upon which higher-level services can be built. In essence, Coyote can be viewed as a microkernel for modern hybrid computing systems, addressing the absence of critical abstractions such as multi-tenancy, virtualization, shared-memory, networking, and more. Coyote incurs small overhead while yielding substantial improvements in both performance and ease of use. Most importantly, it establishes a fundamental framework for the exploration of innovative data processing models and the role of modern heterogeneous platforms in modern computing on a broader scale.

ZUSAMMENFASSUNG

Die Hardware-Landschaft erlebt eine bedeutende Transformation, angetrieben durch steigende Anforderungen an die Datenverarbeitung. Moderne Rechensysteme integrieren vermehrt heterogene Verarbeitungseinheiten, was einen Bruch mit traditionellen Von-Neumann-Architekturen bedeutet. Ein herausragendes Beispiel für diese Diversifizierung sind hybride Rechnerplattformen, die eine Allzweck-CPU mit einem Field-Programmable Gate Array (FPGA) kombinieren und mittlerweile in modernen Rechenzentren und Cloud-Umgebungen zum Standard geworden sind.

Diese Plattformen verbessern die Leistung und Energieeffizienz signifikant über verschiedene Workloads hinweg, dank ihrer bemerkenswerten Flexibilität bei der Bewältigung verschiedenster Verarbeitungsanforderungen. Die Heterogenität dieser modernen Geräte stellt jedoch erhebliche Herausforderungen in Bezug auf Softwarekompatibilität, Programmierung und das Gesamtdesign des Systems dar. Die Navigation durch diese komplexen Plattformen, um das volle Potenzial der vielfältigen Hardware auszuschöpfen, kann anspruchsvoll sein. Insbesondere hinkt die Systemsoftware für diese Geräte hinterher und weist nicht die traditionellen Abstraktionen auf, die in herkömmlichen Betriebssystemen üblich sind.

Diese Arbeit überbrückt die Kluft zwischen moderner heterogener Hardware und der entsprechenden Systemsoftware. Zu diesem Zweck stellen wir Coyote vor, eine umfassende FPGA-”Shell”, die einen Kernsatz von wesentlichen Funktionen und Abstraktionen bietet, auf denen höhere Dienste aufbauen können. Im Wesentlichen kann Coyote als Mikrokernel für moderne hybride Rechensysteme betrachtet werden, der die fehlenden kritischen Abstraktionen wie Multi-Tenancy, Virtualisierung, Shared Memory, Netzwerke und mehr adressiert. Coyote verursacht geringe Overheads und bringt erhebliche Verbesserungen sowohl in Bezug auf Leistung als auch Benutzerfreundlichkeit mit sich. Am wichtigsten ist, dass es einen grundlegenden Rahmen für die Erforschung innovativer Datenverarbeitungsmodelle und die Rolle moderner heterogener Plattformen im modernen Computing im größeren Maßstab schafft.

Моме оси

Acknowledgments

I extend my sincere gratitude to my advisor, Gustavo Alonso, for his invaluable support, abundant advice, and extensive feedback that significantly contributed to the progress of my research throughout my academic path. My appreciation extends to my co-advisor, Timothy Roscoe, for participating in insightful discussions and offering valuable ideas that have significantly impacted my work throughout the years. I'm also thankful for the continuous openness of other members in the System Group, namely Ana Klimović, Theodoros Rekatsinas, and Ghislain Fourny, to participate in discussions on various topics. Additionally, I would like to convey my thanks to Michaela Blott and Lana Josipović for graciously agreeing to be part of my doctoral committee and for the fruitful collaborations we have enjoyed.

I would also like to extend my gratitude to my collaborators from HPE, specifically Kim and Dejan, for the outstanding cooperation over the years that led to great results. I appreciate the valuable learning experiences gained through our joint efforts. My thanks also goes to my collaborators from Microsoft Research, namely Aleksandar, Shane, Juniy, Ho-Cheung, and Igor, for providing an enriching internship experience.

In adition, I wish to express my gratitude to the exceptional researchers and friends with whom I had the privilege of collaborating and spending time with over the years, including Zhenhao, Monica, Johannes, Dan, Dimitrios, Fabio, Thomas, David, Mohsen, Kaan, Abishek, Adam, Simon, Marko, Lazar, Michal, Michael, Andrea, Vasileios, Tom, Wenqi, and many others. It was also a pleasure collaborating with superb students, and I want to extend my thanks especially to Paul, Christian, Mihai, and Karol for their contributions to the research presented in this thesis. A special thank you to our dedicated administrators Nadia, Natasha, Jena, and Simonetta for their consistent assistance in times of need.

Finally, I want to express my heartfelt gratitude to my dearest friends and family, with a special acknowledgment to my parents, for their unwavering support throughout the entirety of my academic journey.

Contents

CHAPTER 1

INTRODUCTION

1.1 General Context

In an era marked by the exponential growth of data, our computational demands are experiencing exponential growth [191]. This coupled with the demise of Moore's law [82] and the end of Dennard Scaling [84] has prompted a significant paradigm shift in the world of computing. The level of effortless scalability in computational performance that was present just a decade or two ago has experienced a drastic decrease in recent times. The classical Reduced Instruction Set Computer (RISC) architectures, once considered scalable and sufficient for most computational needs, now require additional specialized hardware to cope with ever increasing computational demands. As a result, there is a growing trend to explore and implement a variety of heterogeneous architectures involving traditional general-purpose processing units and cutting-edge specialized hardware solutions.

This implies that hardware acceleration is increasingly recognized as a necessity in the realm of modern data centers and cloud computing. The exponential growth in data volumes, the complexity of applications, and the demand for real-time processing have outpaced the capabilities of traditional CPUs. General-purpose CPUs are versatile but fall short when it comes to handling highly parallelized workloads efficiently. This is where dedicated hardware accelerators come into play. They are purpose-built for parallel processing and can significantly enhance the performance of modern workloads. These hardware accelerators come in different flavours, including ASICs, GPUs, and FPGAs. Each with its own set of advantages and applications.

Application-Specific Integrated Circuits (ASICs) represent custom-designed microchips tailored for specific or closely related functions. ASICs are unparalleled in terms of performance and energy efficiency for their intended tasks. However, they are rigidly configured during manufacturing and cannot be reprogrammed for different applications. These extreme levels of specialization make ASICs well-suited for tasks where utmost performance is critical.

The growing importance of machine learning has brought notable examples of ASIC utilization, with the Tensor Processing Unit (TPU) developed by Google [114] being one such case. TPUs serve as dedicated hardware accelerators optimized primarily for matrix operations, a fundamental aspect of many deep learning algorithms. They excel in executing the computations commonly encountered during neural network training and inference, making them a vital component in the machine learning landscape. Still, not every individual or organization can leverage the extensive resources and capabilities possessed by corporate giants like Google, rendering large-scale ASIC projects a formidable task. Graphics Processing Units (GPUs) alleviate some of these complexities by offering added versatility, hence their widespread utilization in modern machine learning. However, numerous application scenarios, particularly in the realm of cloud computing, call for the adaptability to accommodate diverse workloads and evolving demands. This adaptability is an aspect where neither ASICs nor GPUs excel, as these devices exhibit strengths in certain domains of computing while showing severe limitations in others.

Field-Programmable Gate Arrays (FPGAs) offer a unique blend of flexibility and performance. These reconfigurable hardware devices essentially provide a “blank canvas” for developers, allowing them to implement custom circuits dynamically, serving a wide range of processing tasks. In exchange for some of the peak performance seen in ASICs, FPGAs provide a significantly higher degree of generality. Consequently, they prove to be particularly valuable for rapidly prototyping new hardware designs and swiftly adapting to evolving requirements. One of their key advantages lies in the multitude of approaches through which they can seamlessly integrate into the broader system framework.

Their abundant I/O capabilities provide a variety of configuration options. They can operate as conventional CPU-attached accelerators or, in contrast to other accelerators, establish a number of direct connections to external devices and networks. In this role, they establish communication channels between multiple devices, both local and remote, functioning as true “bump-in-the-wire” processors. These “silent” processors seamlessly perform computation on the data as it flows through them, eliminating the need for additional data movement, a key solution to one of the primary bottlenecks in modern data processing. These properties have led modern large scale hyperscalers, such as Amazon and Microsoft to deploy FPGAs in large scale.

The large scale integration of these heterogeneous accelerators coupled to general-purpose CPUs, poses a significant challenge due to the mismatch between long-established abstractions designed for traditional load/store architectures and the intricate nature of these modern accelerators. Instead of having a single CPU with some attached memory and some I/O, these heterogeneous systems encompass a multitude of processing cores, including both traditional and specialized architectures, all engaged in complex interactions.

Each of these processors may maintain their own unique memory address spaces, which might or might not be accessible by other cores through memory-mapped instructions or via bulk DMA transfers. Different address spaces on different cores might even be potentially coherent. Added virtualization layers could exist on top of the attached memories, providing shared-memory models, etc. All these features add layers of complexity, requiring careful consideration.

Regrettably, traditional operating system abstractions are ill-equipped to exploit the full potential of this innovative hardware landscape. Consequently, end users are compelled to engage in low-level "bare-metal" like interactions with the hardware, resulting in developmental complexities, reduced productivity, and limitations on the exploration of high-level concepts and research.

To circumvent the non-existant abstractions, modern FPGA "shells" (essentially operating systems for FPGAs, orchestrating complete operation) limit what can be done on these devices. They hide or outright completely disable features from end users and instead provide very strict computational models. This holds true for Amazon F1 instances [30], that emulate the GPU-style computation, focusing on popular machine learning workloads while overlooking one of the key domains where FPGAs excel, namely, networking. On the contrary, Microsoft's Catapult shell [172] incorporates networking aspects but does not address the potential for these shells to be integrated into a virtualized cloud environment. In both instances, the level of programmability and interaction with these shells remains quite basic, demanding a steep learning curve and hindering their use by individuals lacking significant hardware development experience.

It's evident that there is a need to adjust the current set of abstractions in use (or come up with new ones). We should not only focus on achieving the best possible performance but also on making these systems approachable. While top-notch performance is important, if a system is challenging to program and interact with, it won't gain widespread acceptance and use. To illustrate this point, take into account that in the present day, very few individuals would opt to write code directly in assembly language for general-purpose CPUs, even though it has the potential to achieve peak performance. Instead, we tend to program using higher-level tools and leverage features such as multitasking, multithreading, virtual memory, networking services, and more.

1.2 Thesis Goals

In essence, the evolving landscape of hardware and technology demands that we rethink and refine our approach to abstractions to fully capitalize on the opportunities presented by modern hardware. The most critical abstractions we explore in this thesis are summarized as follows.

- Multitasking is a fundamental concept in traditional operating systems. In a multitasking environment, multiple processes run concurrently, making the most of the system's capabilities. Multitasking is usually absent in modern FPGAs, which leads to underutilization as they grow in size. Our work aims to introduce this missing capability to enable efficient resource sharing of these devices.

- Virtualization is crucial in modern computing for several key reasons. It provides security and flexibility by enabling multiple parallel users to safely distribute and utilize the system's underlying resources. This streamlines the development process by creating isolated environments, abstracting away the complexities of the physical hardware. It is yet another concept missing in modern FPGAs. In this thesis, we implement virtualization across the heterogeneous stack, covering both FPGA-side and CPU-side resources.

- Networking abstractions play a pivotal role in the context of primarily distributed, modern data processing. FPGAs emerge as ideal tools in such scenarios, enabling the reduction of data movement and efficient processing with minimal added costs. We explore how different networking services can be efficiently integrated and utilized in modern FPGAs.

- With the above abstractions implemented, we investigate the development of additional systems that can harness these to improve the performance of modern workloads. We place a particular focus on memory disaggregation and the concept of offloading computation close to data, both of which are prominent trends in modern cloud computing. We showcase how these abstractions, when executed on these heterogeneous platforms, can substantially enhance performance in this context.

The central objective of this thesis is to establish a fundamental set of abstractions that mirror the functionalities of a conventional operating system, tailored specifically for hybrid platforms featuring FPGAs. Our goal is to create a set of tools that have the potential to enhance productivity on these cutting-edge platforms for a wide range of users. Additionally, we aim to reduce the obstacles to exploring these platforms, especially for users with limited experience.

1.3 Contributions & Structure

The primary contribution of this thesis is the introduction of *Coyote*, a holistic framework that provides a range of hardware and software abstractions for contemporary heterogeneous systems. This framework is extensively detailed throughout the thesis, beginning from the foundational layers of the stack and advancing to the software interfaces designed for end-users. The culmination of this work is the development of *Farview*, a practical system build on top of Coyote that demonstrates how heterogeneous hardware can effectively support current trends in memory disaggregation. These contributions are summarized as follows.

- In Chapter 2 we establish the basis for the entire thesis by addressing the fundamental question: "How relevant are traditional operating system abstractions in the context of an FPGA within a hybrid system?" This chapter lays the groundwork for the in-depth exploration of Coyote's implementation.
- In Chapter 3, we explore the core (kernel) of Coyote. This chapter emphasizes the critical interaction between the host CPU and the FPGA, serving as the foundation upon which additional abstractions and services within the system are constructed. We present the interfaces that render Coyote adaptable to different platforms and substantiate this by successfully porting it to a custom research computer. Additionally, We investigate hierarchical dynamic reconfiguration, providing Coyote with a flexibility that surpasses that of other state-of-the-art FPGA shells.
- In Chapter 4, we explore the configurable "shell" layer that provides system-wide abstractions and services within Coyote. We begin by concentrating on the memory architecture among devices and the development of the encompassing shared virtual memory model. Subsequently, we shift our attention to the network structures, where we analyze both the TCP/IP and RDMA network stacks and their integration into a multi-tenant environment.
- In Chapter 5, we introduce the standard execution environment in Coyote, facilitated by a series of clearly defined interfaces through which users can seamlessly interact with all Coyote services, whether they are local or remote. We also explore the software layer in Coyote, beginning with software virtualization, which offers the capability to fully encapsulate Coyote's tenants in virtual machines. We then examine the runtime abstractions and end-user software application interface. We conclude this chapter with a practical application that can leverage Coyote's abstractions.

- In Chapter 6, we utilize Coyote as a foundation to construct supplementary systems. In this section we introduce *Farview*, a disaggregated memory solution designed for modern databases, which functions as a remote buffer cache with operator offloading capabilities. Farview illustrates how smart disaggregated memory can serve as a practical alternative for databases deployed in cloud environments.

A portion of the work used for this thesis has been published and is included here for reference:

- **Do OS Abstractions Make Sense in FPGAs?** *by D. Korolija, T. Roscoe, G. Alonso, in the 14th USENIX Symposium on Operating Systems Design and Implementation*
- **Farview: Disaggregated Memory with Operator Off-loading for Database Engines** *by D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. Milojičić, G. Alonso in the 12th Conference on Innovative Data Systems Research*
- **LynX: A Flexible FPGA Virtualization Framework for Heterogeneous Systems** *by D. Korolija, T. Roscoe, G. Alonso, in 10th Workshop on Systems for Post-Moore Architectures*
- **Enzian: an open, general, CPU/FPGA platform for OS research** *by D. Cock, M. Giardino, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, A. Ramdas, A. Turowski, G. Alonso, T. Roscoe, in the Architectural Support for Programming Languages and Operating Systems 2022*
- **Data Processing with FPGAs on Modern Architectures** *by W. Jiang, D. Korolija, G. Alonso, in the Companion of the 2023 International Conference on Management of Data*
- **Serverless FPGA: Work-In-Progress** *by F. Maschi, D. Korolija, G. Alonso, in the 1st Workshop on SErverless Systems, Applications and MEthodologies*

Chapter 2

Do OS Abstractions Make Sense on FPGAs?

As computational demands continue to grow, hybrid computing systems that tightly integrate CPUs and FPGAs are increasingly common in modern data center and cloud settings. The combination of these heterogeneous devices can yield significant performance advantages and greater energy efficiency compared to conventional general-purpose architectures. Moreover, they provide significantly greater flexibility when compared to conventional ASIC accelerators, which constitutes one of their primary strengths.

However, it is crucial to recognize the existing challenges involved. In general, working with these systems, including development, deployment, and interaction, can often prove to be highly complex and cumbersome. To make the most of the performance and flexibility of these devices, the overall system interaction frequently involves the utilization of a low-level application programming interface (API) that closely resembles that used in traditional bare-metal programming. Inherent lack of support for higher-level abstractions, which we typically rely on in traditional operating systems, such as processes, multitasking, memory virtualization, and various accompanying services, becomes a significant constraint as FPGAs grow in size and capability. This limitation significantly reduces the overall productivity on these devices.

This has prompted recent efforts in research to replicate a portion of the traditional operating system execution environment on FPGAs, encompassing aspects such as processes, threads, scheduling, virtualization, and more. Yet, in these efforts, the predominant focus is on individual abstractions, whereas what makes these abstractions so challenging and difficult to explore lies in the interaction betwen them (Figure 2.1). As an example, virtual memory depends on the underlining physical memory management within the system. Services, which play a vital role

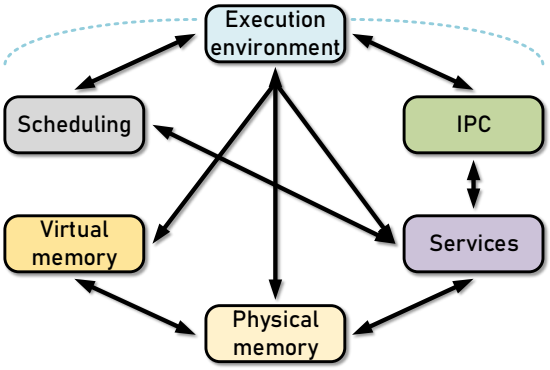

Figure 2.1: Interdependencies among various conventional OS abstractions.

in simplifying the user's experience, can, in turn, be dependent on this virtual memory. The scheduling of user applications needs to take into account all the stateful services that exist and must be executed without placing trust in the running user processes, each of which should also be kept isolated from one another, and so forth.

Given the above, in this chapter, we give an initial introduction to *Coyote*, the system we built to examine to what extent do traditional OS abstractions remain relevant when applied to an FPGA as part of a hybrid system, especially when considering them as an integrated whole, as they would be in a traditional operating system.

2.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are increasingly being adopted as a standard technology in datacenters and by cloud providers [30, 26, 62, 45]. Hardware acceleration is thus becoming increasingly relevant for a variety of use cases, e.g., machine learning workloads [118, 201, 166], data processing [128, 67, 144], high performance computing [56, 53] and networking [91, 185]. Reconfigurable platforms in these domains offer increased flexibility and lower power consumption compared to ASICs or GPUs for many applications, while retaining the performance (e.g. [43, 92, 137, 200, 129]). Despite their benefits, these platforms (due to their origins in embedded systems and prototyping), still present major challenges in terms of programming, deployment, and secure management. As a result, along with much research into making FPGAs easier to program [7, 159, 197, 203, 46, 180, 47], considerable recent work applied ideas from operating systems design and implementation to resource allocation, sharing, isolation, and management of an FPGA-centric computer.

So far, this work has been piecemeal, primarily focusing on a particular aspect of functionality in isolation, e.g. Feniks [212] targets solely FPGA access to I/O peripherals, Optimus [146] provides access to a host's virtual memory via address translation, AmorphOS [122] prioritizes the scheduling of user applications as its central objective, etc. All these yield substantial incremental improvements over existing solutions and constitute the current state of the art.

At the same time, what makes good OS design so challenging is the close interaction in the kernel between *all* the functionality. Virtual memory without support for multiple applications (multi-tenancy) or strong isolation between them is of limited use. Virtualizing hardware devices without providing virtual addressing and creating a common execution enviroment that abstracts the hardware leaves most of the problem unsolved. An FPGA scheduler that cannot exploit the ability to dynamically reconfigure parts of the chip or/and cannot properly capture the complete state of the application running within the FPGA has a limited shelf-life, and so on. Moreover, in contrast to traditional operating systems that take advantage of the underlying general-purpose hardware, integrating these abstractions into a hybrid system with reconfigurable fabric introduces significant additional challenges. Questions arise about how these abstractions are mapped to the physical hardware, their physical placement, whether the physical timing requirements can be met, among many other considerations.

Therefore, we step back to ask the question: to what extent can (or should) traditional OS concepts (processes, virtual memory, etc.) be usefully translated to an FPGA? What happens when they are? To answer this question, we need to adopt a comprehensive, holistic approach and think about complete functionality, rather than sticking to particular aspects of an OS or supporting only limited FPGA features.

To this end, we developed *Coyote*, combining a coherent set of OS abstractions in a single unified runtime for FPGA-based applications. Coyote serves as an FPGA shell, assuming control over the entire FPGA fabric as well as all the connections to the external environment. A suitable analogy for Coyote within the domain of operating systems would be to liken it to a microkernel. It provides the core set of essential functions and features on which other services can be based: a uniform execution environment and portability layer, virtualization, physical memory management, communication, spatial and temporal scheduling, networking, and an analog of software processes or tasks for various custom user logic and applications. Coyote helps us to examine a range of OS abstractions and to provide a critical assessment of how each might map to an FPGA, especially in the context of its interaction with other abstractions.

2.2 Foundations

We begin this section by assessing the hardware architecture commonly encountered in hybrid platforms deployed within modern data centers and cloud systems. Our focus will be on the organization of the *shells* running on these platforms. These shells constitute the system software layer responsible for exposing the functionality of these platforms to end users.

These hybrid platforms predominantly feature a combination of a conventional server-grade CPU and a high-performance FPGA. Typically, these two components are connected via a conventional peripheral bus like PCIe (shown in Figure 2.2), as observed in several AMD and Intel data center cards [12, 6, 5]. Because of the nature and complexity of hardware development, this CPU-FPGA interconnect serves a dual purpose. It not only facilitates data transfers during different processing stages but also handles the majority of the control and management tasks on these platforms (significantly simpler to accomplish on a general-purpose CPU). As a result, it assumes a pivotal role in the overall design decisions within the system.

Different configurations with conventional PCIe interconnect are prevalent in modern hyperscaler cloud shells offered by companies such as Amazon [30] and Alibaba [26], using AMD commodity hardware (Figure 2.2). Microsoft, similarly, employs Catapult [62], their proprietary, internally developed combination of hardware and shell (Figure 2.4). In these cloud configurations, the model of computation is primarily centered around the bulk-oriented data movement between the host CPU and the FPGA accelerator. However, due to the limited abstractions in the overall systems software support, these frameworks offer limited flexibility in terms of a computation model. They typically adhere to a rigid GPU-style of computation, where data must first be copied to the FPGA's local memory before starting the computation, and subsequently copied back upon completion. Additionally, most of these platforms are completely *host-centric*, relying fully on the CPU to manage complete transactions. As a direct result, these limitations in the compute model add significant overheads.

More recently, efforts have been made to mitigate these limitations, especially for fine-grained workloads, by connecting FPGAs through cache-coherent interconnects such as CXL [73], CCIX [63], OpenCAPI [188], or Intel UPI [19]. In the research community, an example of such a system is an ongoing initiative with Enzian (shown in Figure 2.5), a research computer built at ETH Zurich [101], utilizing the proprietary ECI (Enzian Cache-coherency Interface) protocol [28, 70]. These interconnects, coupled with cache coherency, deliver reduced latency and significantly enhance the "bond" between the host CPU and FPGA devices. Cache coherency,

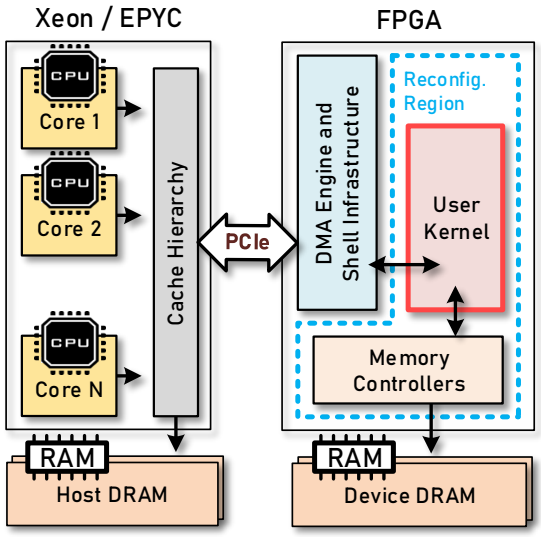

Figure 2.2: Traditional PCIe based platform.

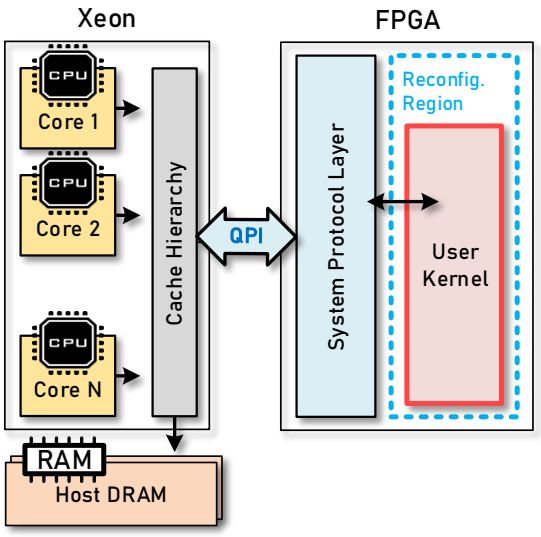

Figure 2.3: Intel HARP v1.

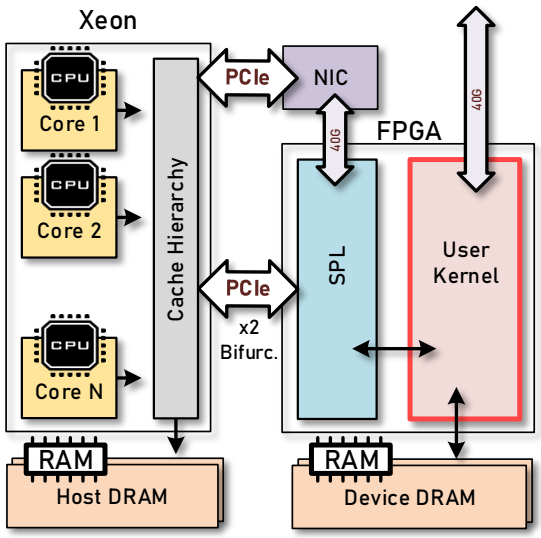

Figure 2.4: Microsoft Catapult.

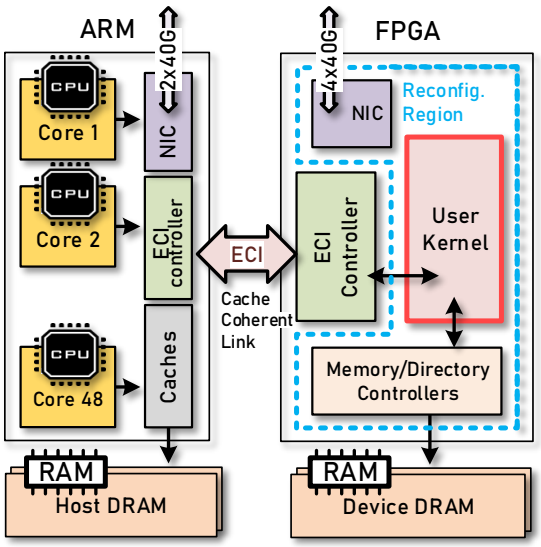

Figure 2.5: Enzian v3.

in particular, unlocks some interesting use cases that can yield performance benefits in various domains, such as RPC or memory management [132, 61, 60].

Nevertheless, a significant proportion of data flow latency-insensitive workloads (such as analytical database processing or some machine learning algorithms) which involve processing

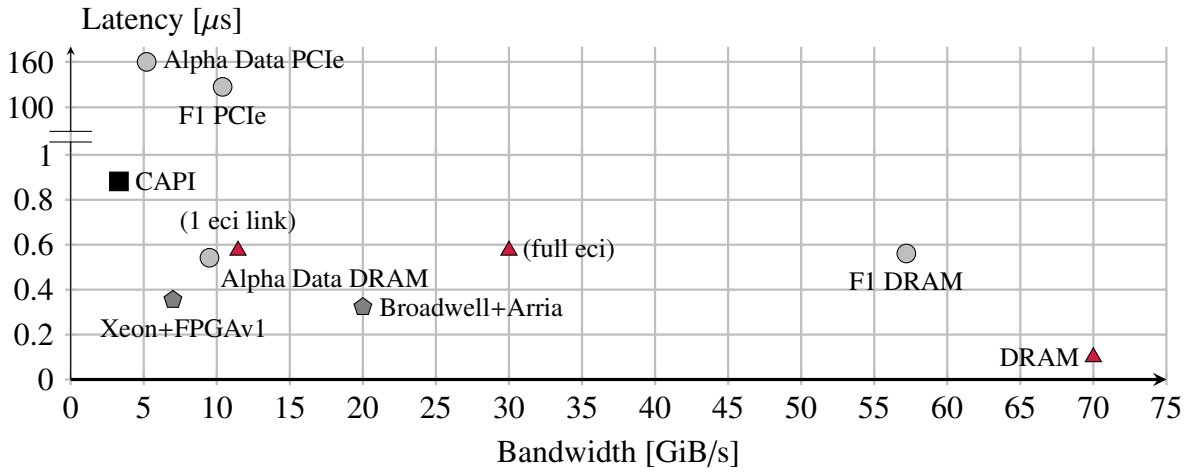

Figure 2.6: Interconnect throughput and latency performance comparison (adapted from Enzian paper published at ASPLOS '22 [70].

on streamed data elements within deep pipelines, may not experience substantial benefits from these cache-coherent interconnects. This can be inferred from the experiments done on CXL with Pond [135]. Instead, these workloads are more likely to derive significant advantages from the reliability and comprehensive system support provided by a more conventional peripheral bus. It's worth noting that FPGAs in these configurations have also demonstrated proficiency in handling these types of workloads effectively. While there certainly are latency-sensitive workloads [135], particularly applicable to graph processing algorithms, it remains uncertain to which extent are FPGAs well-suited for these type of fine-grained tasks.

Besides the host interconnect, another crucial and distinctive feature of these hybrid systems incorporating FPGAs is their effortless integration with abundant I/O resources. Usually, this I/O is directly linked to the FPGA fabric, enabling applications within the FPGA to interact with this I/O with minimal overhead. This capability represents a fundamental advantage of FPGAs, as it facilitates the swift transfer of data from the host, memory, storage, network, etc., directly to the processing elements with minimal overhead, serving as the primary source of performance advantages that these devices can yield.

However, a significant challenge arises from the fact that various types of I/O, including high-speed networking, external off-chip memory, storage devices, and external accelerators, each come with their own unique low-level interfaces that often require substantial system software support. Given that FPGAs usually arrive with minimal shells lacking features reminiscent of those found in conventional operating systems, a substantial portion of this I/O often remains

underutilized. As an illustrative example, consider the current state-of-the-art AMD Alveo data center cards [12], which boast high-performance 100 Gbps networking interfaces. Surprisingly, the default vendor shell for these devices [207] does not provide any networking support by default, thus leaving this valuable I/O capability untapped. A comparable scenario could be noted with the competing platform, where earlier versions of Intel's HARP [162, 167], shown in Figure 2.3, provided extremely limited support for FPGA side, off-chip memory components.

A critical component in modern computing is the high-performance network interface. Contemporary FPGAs come equipped with high-speed transceivers capable of handling the data rates commonly seen in today's data center networks. This means that FPGAs can seamlessly integrate with the network and directly process incoming and outgoing network packets. Additionally, as mentioned, FPGAs are connected to various other resources, including host CPU memory through PCIe, local FPGA memory, storage, and other network ports. This connectivity allows FPGAs to function as smart Network Interface Cards (NICs) and perform hardware acceleration tasks with minimal impact on data movement. Essentially, FPGAs act as "bump-in-the-wire-processors", making them well-suited for various streaming workloads.

Microsoft Catapult project extensively explored the significance of this capability. It aimed to accelerate network processing for Microsoft's cloud workloads, leveraging FPGAs. One example of a workload that benefited significantly from this acceleration was the page-rank algorithm [62] used in production for Bing searches. Another noteworthy project, BrainWave, demonstrated the effectiveness of FPGAs in large-scale graph neural network inference [68].

Catapult platform stands out among data center heterogeneous platforms due to its unrestricted user access to networking resources. While there are dedicated smart NICs equipped with reconfigurable hardware, they often feature smaller FPGAs and therefore place less emphasis on reconfigurable hardware. Both these dedicated smart NICs and Catapult share a common feature in fully hardened NICs situated outside the FPGA fabric. This design choice, while enhancing NIC performance, imposes limitations on the extent of system-level exploration one can do with these smart NICs as large portions of the system are hardened.

One of the primary advantages of FPGAs is their inherent flexibility, which is derived from their reconfigurable nature. This adaptability is most effectively harnessed through technologies like partial reconfiguration [198]. With partial reconfiguration, specific sections of the FPGA logic can be exchanged with alternative logic "on-the-fly", during runtime, without impacting the functionality of other parts of the FPGA logic. This capability enables users to seamlessly swap accelerators, representing a significant advantage of these devices over more traditional ASIC accelerators.

Regrettably, large-scale heterogeneous platforms rarely offer robust support for partial reconfiguration. While these platforms often employ it for loading the accelerators, there is minimal effort dedicated to providing users with fine-grained control over the reconfiguration process, as it remains largely concealed.

These large-scale cloud platforms consistently offer only single-tenant environments, which poses a significant drawback. This is especially problematic as FPGAs continue to grow in size, and a substantial number of accelerators can be accommodated with just a fraction of the available FPGA resources [126]. Consequently, significant portions of the FPGA fabric remain underutilized, leading to adverse effects on the overall cost, performance and efficiency.

As evident from the information above, the currently available shells and platforms consistently have certain limitations in terms of features and functionalities. Consequently, they offer restricted environments for both research and the deployment of various types of applications. For instance, while a system like Catapult may excel in the context of distributed processing across the network, it may fall short in other areas, lacking support for multi-tenancy, reconfiguration, or any form of virtualization. In general, these frameworks lack fundamental OS abstractions such as multi-tenancy, virtualization, memory management, networking services, etc. This limitation restricts their utility, making them highly specialized and somewhat counterproductive for leveraging the inherent flexibility of such hardware.

Moreover, these frameworks come with intricate interfaces, rely on proprietary hardware and software IP, and entail a steep learning curve for developers. System abstractions present on one platform will typically be absent or markedly different when transitioning to other platforms. Thus achieving portability across these platforms becomes a challenging task, with minimal opportunities for cross-compatibility.

We developed *Coyote* to overcome the mentioned limitations. Currently, Coyote is compatible with a wide array of AMD Alveo data center cards, including models such as Alveo-U250, Alveo-U280, Alveo-U55C, Alveo-U50D, and Alveo-U200 [8, 9, 10, 15, 14]. In addition to the aforementioned compatibility with AMD Alveo data center cards, Coyote is also capable of running seamlessly on Enzian [70]. This real-world example serves as concrete evidence of the essential portability aspect of the entire system.

A central component of Coyote's design is the establishment of a multi-tenant environment, akin to the structure of a traditional microkernel. Significantly, this environment additionally provides a spectrum of services that are available to all tenants. These services, including virtualization, memory and network stacks, play a pivotal role in boosting productivity on these

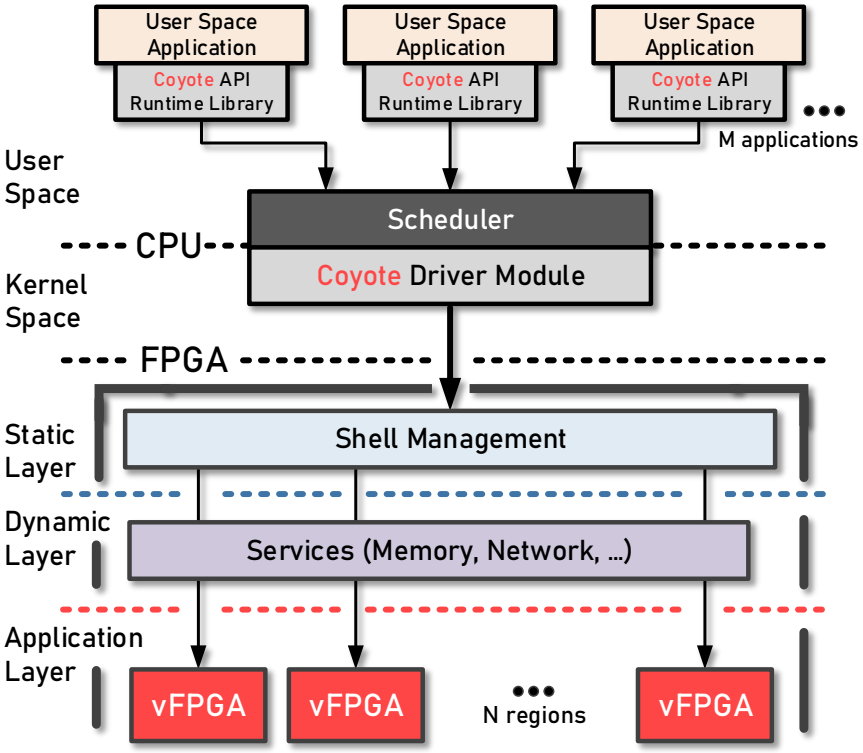

Figure 2.7: Coyote high level overview of the system architecture.

devices. Importantly, we have deliberately steered clear of any design choices that could hinder the utilization of modern FPGA capabilities, such as dynamic partial reconfiguration of multiple regions or valuable “hard” on-chip functions.

A high-level overview of Coyote is depicted in Figure 2.7. The framework architecture is naturally divided into two primary components. The first component resides in the ”hardware” realm and runs on the FPGA, while the second component operates on the host CPU, as part of the OS and support libraries.

2.2.1 The Hardware “Split”

As is customary in the cutting-edge shell designs, dynamic reconfiguration of the FPGA induces a further split of the hardware component into a “static region”, configured at boot, and a “dynamic region”, containing subregions (vFPGAs), each of which may be changed on the fly. This split exists (often in simplified form) in all FPGA datacenter deployments.

Within and between regions, it is customary for hardware components to interact via interconnects like AXI [145]. The AXI (Advanced eXtensible Interface) protocol stands as a widely embraced and standardized interface within the areas of digital design and computer engineering. Its primary function is to enable seamless communication among diverse hardware components residing within SoCs, FPGAs, or ASICs. This protocol is crafted to enhance performance, scalability, and the ease of integration within digital systems.

Within the context of Coyote, both full AXI4 memory-mapped interfaces and AXI4 streaming protocol (a specialized iteration of AXI optimized for continuous data flows) are employed. The streaming protocol, which uses a simple handshaking logic, is particularly well suited for the creation of intefaces exposed to the end user applications. This protocol represents the preferred choice within Coyote. We'll postpone deliberating on this selection for a later time (Section 3.1.1).

2.2.1.1 Static Layer

The FPGA static layer must contain the functionality required to reconfigure the dynamic layer and communicate with the CPU's OS. This layer needs to contain only the essential features which support the operation of user applications and this is indeed the case for most of the vendor shells out there.

In Coyote, the static layer always contains logic to partially reconfigure portions of the dynamic layer, communicate with the host machine (a DMA copy engine), and to divide the dynamic region into a set of *virtual FPGAs* ("vFPGAs"), each of which has an interface mapped into the physical address space of the host CPU.

Considering that Coyote is designed for data center applications and is typically deployed within large clusters, it's essential to highlight that the static layer remains in a constant state of being "online". This continuous availability ensures that logic within the FPGA can be adapted and modified without the necessity of taking the entire system offline or resorting to procedures like cold reboots.

It's important to note that this layer is the sole layer in Coyote that is inherently platform-dependent. Currently employed approach is to maintain it internally under strict version control for all platforms. However, given that Coyote is an open-source research project, users have the flexibility to experiment and explore various configurations and modifications within this layer at any given time (at their own risk).

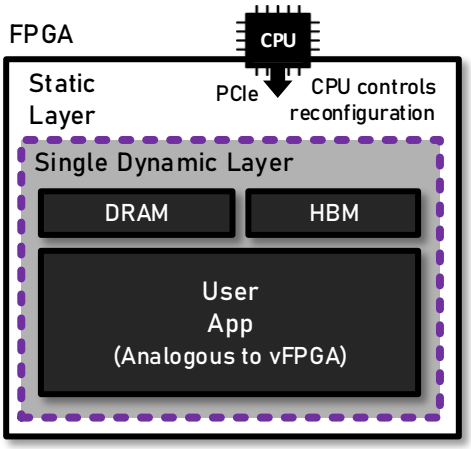

Figure 2.8: Amazon F1 FPGA shell reconfiguration.

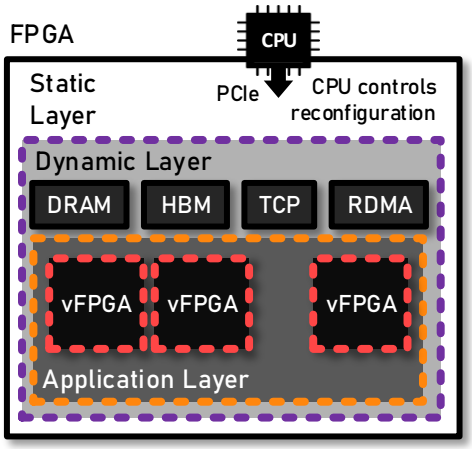

Figure 2.9: Coyote FPGA shell reconfiguration.

However, what truly sets Coyote apart from similar systems is its dynamic layer. Its uniqueness is found in how multi-tenancy and FPGA-based services come together and interact within this layer.

2.2.1.2 Dynamic (Service) Layer

The dynamic region is the basic mechanism for time-division multiplexing of the FPGA resources. Modern FPGAs allow selective portions of this region to be reconfigured independently at any time.

Most deployed systems (e.g. F1 [30] and Intel's HARP [162]) dedicate this region to a single application, and reprogram it only rarely with large overheads (e.g. when an associated virtual machine on the host is booted up).

Space (chip area, logic blocks, wires, etc.) remains a scarce resource on FPGAs, and unlike OS resources such as CPU time and virtual memory, it is hard to make it "elastic" through virtualization. Moreover, services like memory stacks (HBM, DRAM) and networks stacks (TCP/IP, RDMA) contain large complexity and use significant FPGA resources, thus it is is important to make these components optional so they can be ommited if not needed. Additionally, the number of tenants can also change dynamically, thus all of these modules need to be part of the dynamic fabric which can be configured on-the-fly to best utilize the available resources.

Coyote supports multi-tenancy as its core abstraction, further extended by the inclusion of system-wide services. Each tenant within the system can be reconfigured independently without impacting other tenants. This concept closely aligns with the notion of processes in operating systems. This stands in sharp contrast to traditional single-tenant shells, which always allocate the dynamic region to a single application.

Coyote introduces added complexity through its system-wide services. These services are utilized collectively by all tenants and cannot be merged within the same dynamic regions as the applications. The creation of distinct dynamic regions for each service is also not a viable solution, as it would lead to notable congestion and floorplanning challenges (dynamic regions come with strict resource and floorplanning requirements). Moreover, these services cannot be interchanged as freely as applications and typically operate at a lower level within the notion of the operating systems.

To address this complexity, Coyote employs a solution by using *hierarchical dynamic reconfiguration*. This approach allows for the creation of layers within layers of reconfigurable regions, granting the flexibility to not only swap dynamic user applications but also the services that serve these applications. A comprehensive explanation of the hierarchical reconfiguration approach in Coyote will be provided later on in Chapter 3.3. Both the service layer (memory and network stacks) and the application layer are integral components of the dynamic layer within Coyote.

2.2.1.3 Application Layer

At the application layer, Coyote, much like other recent systems [212, 86, 64, 210], offers flexible spatial and temporal multiplexing. The application layer is divided into independent *virtual FPGAs* (vFPGAs), enabling the concurrent execution of multiple applications and their seamless switching in and out. Since the entire layer is part of the overarching dynamic layer, the number of these vFPGAs is a compile-time parameter determined by the shell. Different versions of the dynamic layer loaded during runtime can accommodate varying numbers of vFPGAs. This adaptability can also be leveraged for efficient resizing and scheduling of user applications based on current occupancy.

A novel aspect of Coyote is its division of each vFPGA into two components: the *user logic* and the *user wrapper*. The user logic consists of a bitstream that is entirely synthesized by a Coyote user, subsequently validated by the system, and contains both trusted and untrusted segments. It is crucial to emphasize that the user logic is regarded as untrusted, and precautions are implemented to prevent users from potentially engaging in malicious actions that could impact

the system adversely. Furthermore, the Coyote management layer maintains the capability to remove the entire state of the user logic at any given moment, if deemed necessary.

It is tempting to draw an analogy between the structure of Coyote and a microkernel model of an OS, consisting of the kernel (the static layer), services (the service layer), system libraries (the dynamic layer), applications code (the application layer) and user processes (vFPGAs). However, this would not be entirely correct. For example, the user wrappers in the dynamic layer form part of a trusted computing base (TCB), whereas system libraries in a microkernel do not. Nonetheless, this analogy retains significance and can serve as a simple means to visualize the layers in Coyote.

The user wrapper is part of Coyote, and both sandboxes user logic and imporantly provides a set of standard interfaces to the rest of the system (in FPGA terms, partition pins are inserted by the reconfiguration tool locking all the boundary interface signals in the fabric). This incurs a cost in chip area usage (same applies to the above service/dynamic layer), but the benefit is that Coyote pushes the "portability layer" for FPGA applications up to the language level: an application written for Coyote can, given sufficient resources, be synthesized to run on any Coyote FPGA. In contrast, with native FPGA development at present, code is rarely portable between device models (or even, in some cases, revisions of the same model).

We will dive into a more in-depth examination of the *Unified Logic interface* available to users in a later Section 5.1. At this point, it's worth noting that this interface primarily comprises straightforward AXI streams, offering significant flexibility in programming models. Due to the straightforward nature of these streaming interfaces and their broad range support, developers in Coyote can create applications using a variety of language models, including traditional Hardware Description Languages (HDL), various forms of High Level Synthesis (HLS), or even a blend of these and other programming languages.

Traditional hardware description languages (HDLs) such as Verilog and VHDL function at the Register Transfer Level (RTL), allowing the description of circuits using either behavioral or structural representations. This facilitates fine-tuning of designs to achieve optimization in both resource utilization and performance and thus most of Coyote's infrastructure is in HDL.

HDLs offer an excellent abstraction for hardware development demanding utmost precision. However, it's crucial to acknowledge that, especially when working at the application level, achieving efficient utilization can be challenging due to the need for a profound grasp of the microarchitectural specifics, spatial characteristics, and timing constraints inherent to the target device, which can potentially hinder productivity.

Listing 2.1: Top Level HLS code of the *Hyper-Log-Log* operator in Coyote.

```
void hyper_log_log (
    // Coyote Host Streams
    hls::stream<ap_axiu<AXI_DATA_BITS, 0, PID_BITS, 0> >& axis_host_sink,
    hls::stream<ap_axiu<AXI_DATA_BITS, 0, PID_BITS, 0> >& axis_host_src,

    // Coyote CSR
    ap_uint<64> axi_ctrl
) {
    #pragma HLS DATAFLOW  disable_start_propagation
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register port=axis_host_sink name=s_axis_host_sink
    #pragma HLS INTERFACE axis register port=axis_host_src name=m_axis_host_src
    #pragma HLS INTERFACE s_axilite port=axi_ctrl bundle=control

    // Hyper Log Log Kernel
    hyper_log_log_compute(s_axis_host_sink, m_axis_host_src, axi_ctrl);
}
```

These challenges associated with using HDLs are widely recognized, and as a result, considerable effort has been dedicated to the development of high-level synthesis (HLS) languages to simplify FPGA programming. Modern tools like Vitis [207] and Intel HLS [109], for example, employ the syntax of conventional programming languages such as C. These tools abstract the intricacies of hardware, apply traditional software compiler optimization techniques, and ultimately transform a program into a hardware representation through allocation and scheduling processes. For these productivity reasons, the decision to use streaming interfaces was significant because they enjoy strong support within HLS. This choice enables Coyote and its interfaces to seamlessly support HLS kernels without requiring additional modifications. Listing 2.1 shows the top level of the Hyper-Log-Log operator written in HLS for Coyote.

Unfortunately, extracting parallelism from sequential HLS code is a challenging task, necessitating users of such compilers to provide extra input and hints to the compiler to achieve a reasonable level of result quality. Tools like Xilinx's HLS or Intel's OpenCL heavily rely on programmers adding "pragmas" [207] to specify where and how to parallelize the final design. Pragmas enable developers to annotate code with additional information that compilers or HLS tools can utilize, such as marking loops as parallelizable or pipelining specific code sections, but add a layer of complexity to otherwise very C like code.

While HLS can undoubtedly enhance productivity in hardware development, it's worth noting that significant portions of the code written in these languages deviate from the conventions of traditional C code and frequently adhere to typical hardware programming patterns [138], which

can additionally make the code difficult to debug [85].

Therefore, while HLS is a valuable tool, it may not provide an ultimate solution, and having the ability to combine both traditional HDL and HLS support is a preferable option. The interface choice in Coyote facilitates this possibility.

2.2.2 Host System Software

In a hybrid system, the host operating system must clearly be aware of the FPGA environment, and also provide suitable and safe abstractions to user application code running on the CPU for interacting with user logic on the FPGA.

Beyond this, however, there is a fundamental tradeoff between how much management of FPGA resources is performed on the FPGA itself (by a combination of static region logic and dynamic functionality) and how much is implemented by system software on the CPU. Offloading FPGA management functionality to the CPU and OS frees up valuable space on the FPGA, and allows much more policy flexibility than could be reasonably implemented in logic. Hovewer a functionality that is on a critical path can lead to degraded performance and/or loss of predictability in response time (often a key attribute of hardware solutions). In some ways this mirrors the traditional OS tradeoff between kernel-mode and user-space implementation, but the contrast is more stark.

Coyote maximizes the FPGA area allocated for user logic by offloading many non-essential functionalities onto the host CPU's operating system. The software component of Coyote comprises a kernel *driver* that handles most of the core operations (currently designed for Linux), a *runtime manager* process, and user application code.

2.2.2.1 Kernel Driver

During startup, the driver reads the configuration of the static layer on the FPGA and establishes the initial data structures required to support the system's operation within the cluster. At this point, the driver can be directed to load different versions of user supplied *dynamic shells* with various configurations of virtual FPGAs (vFPGAs) and associated services. Subsequently, it creates *character file devices* for each of these vFPGAs, which are then exposed in the Linux kernel. In addition to this a separate character device is created for reconfiguration purposes.

These character devices constitute the main interface between the Coyote driver and the user space applications. Communication between the user space and the driver is achieved through system calls, with a significant portion involving the use of functions like *mmap* and *ioctl*.

The *mmap* calls are employed to directly map various memory locations from hardware directly into user space, allowing Coyote to bypass the kernel in many instances. These direct memory mappings offer substantial performance advantages for a wide range of tasks, including any kernel invocations, initiating data transfers, and launching network calls such as Remote Direct Memory Access (RDMA), among others. This approach bears similarities to DPDK [11] (Data Plane Development Kit) in that it shifts a substantial portion of functionality (in DPDK's case typically related to networking), into user space, circumventing the kernel and thereby improving efficiency.

The *ioctl* calls function as a method for general communication between user space libraries and the Coyote kernel driver. This interface has proven to be reliable and well-established in controlling device drivers. It is also extensible as it allows easy incorporation of new commands. Importantly, it can be utilized through character devices and sockets, which are the main file descriptors used within the Coyote framework.

The *sysfs* file system serves as the default status and debugging tool provided by Coyote. It enables the exposure of various hardware parameters for transparent configuration. Additionally, it provides access to various live system information, facilitating simple debugging and monitoring processes.

The kernel driver also manages the broader control plane communication with the FPGA. This includes tasks like reconfiguring a vFPGA and establishing memory mappings to enable direct interaction between application code and a vFPGA. Additionally, the driver has several other duties, including dynamic memory allocation, processing user notifications from hardware, managing page faults, addressing TLB misses, overseeing memory migrations, and more.

The Coyote driver also operates as a *hypervisor*, providing the system with the capability to run virtual machines on top of individual hardware processes (vFPGAs). This represents the ultimate layer of hardware abstraction and isolation that Coyote provides.

2.2.2.2 Runtime Manager and API

Coyote's runtime manager operates within user space. This component of the system software takes on the role of coordinating interactions among numerous concurrent software users and

the corresponding multi-tenant hardware in the FPGA. It is responsible for scheduling user-submitted tasks. Thus, it manages partial reconfigurations and handles all communication with the reconfiguration character device. The decision to implement this layer in user space, as opposed to kernel space, is grounded in two key advantages: 1) it allows us to bypass potential kernel overhead, and 2) it leverages the extensive array of user space libraries to simplify the implementation of scheduling tasks.

In Coyote, the Application Programming Interface (API) offers additional abstractions aimed at harnessing the potential for parallelization on both the FPGA and CPU sides. The central abstraction provided by this API is the *Coyote thread*. This object represents a singular thread of execution within the hardware pipelines in vFPGAs. In this way, it can be described as a hardware thread running within hardware processes (vFPGAs). Each Coyote thread is also linked to its corresponding software thread of execution (standard POSIX threads [1]). Users have the flexibility to submit their arbitrary tasks to these threads and distribute them between hardware and software execution as desired.

2.3 OS abstractions on an FPGA

Now that we have covered the fundamental aspects of Coyote, we will go into a discussion about traditional OS abstractions and how they can potentially be applied to heterogeneous devices. In the upcoming chapters, we will examine the detailed implementation of Coyote's abstractions. For each OS abstraction under consideration, we will begin by reviewing its role in a conventional OS designed for homogeneous multicore machines. We will then explore what fundamentally differs in an FPGA environment and how this impacts the design decisions when creating an equivalent abstraction on the FPGA.

2.3.1 Processes, Threads, and Tasks

Most operating systems offer fundamental abstractions for efficiently managing and virtualizing processor resources. These abstractions typically revolve around processes, threads, and/or tasks, and they play a pivotal role in enhancing the user experience. While precise definitions can differ between operating systems, a thread typically represents an ongoing execution of a set of instructions on a single virtual processor. A task, represents a discrete unit of computa-

tional work that can be assigned to a CPU core. Meanwhile, a process is typically a collection of threads that share a common address space, isolated from other processes, to which CPU resources are allocated.

The hardware mechanisms underlying these abstractions are basically the ability of the processor to context switch, and be preempted by an interrupt or trap.

Such abstractions can be readily adapted to architectures like GPUs, which retain the notion of a hardware thread, albeit with a very different degree of parallelism. GPU drivers for modern OSes attempt to extend the process abstraction of the host CPU to the GPU, although in a somewhat limited form [51], and this is the foundation for programming models like CUDA and OpenCL. The task abstraction has also been successfully deployed on GPUs [178].

What's Different on an FPGA?

Resource multiplexing on FPGAs is fundamentally different, since there is no hardware entity corresponding to a "processor", "core", or "hardware thread" on which to base an abstraction aimed at multiplexing processing resources.

Instead, two basic mechanisms (Figure 2.10) available on the FPGA for multiplexing compute resources between different users are:

- **Spatial partitioning** of the application logic within the FPGA fabric. This is also one of the primary techniques for optimizing FPGA utilization and performance.
- **Partial reconfiguration** where certain portions of the FPGA logic are swapped during runtime, all while leaving the rest of the fabric unaffected in its operation.

While it is true that a popular programming technique for FPGAs involves implementing a custom application-specific processor (typically some VLIW-based architecture), this is not intended to be multiplexed or scheduled. The analogue of these custom cores in the software world is more that of a library or bytecode interpreter that lives entirely within the process abstraction.

The trivial approach here is to dedicate the entire FPGA to a single application, and indeed in embedded systems this is the norm. A more flexible approach allows more than one application to use the FPGA at a time. The static region of the FPGA contains enough logic to swap one application out for another, but otherwise the chip is dedicated to an application for long periods. This is the model adopted by Amazon F1 and, indeed, almost all other commerically deployed systems.

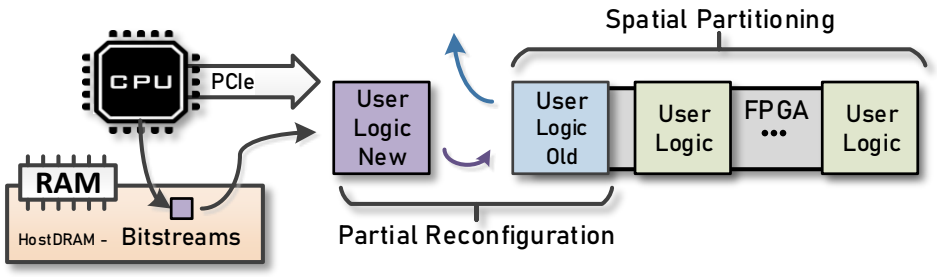

Figure 2.10: Mechanisms for multiplexing within FPGAs

An alternative approach proposed in research systems (e.g. [210, 212, 146] and others), is to statically partition the FPGA resources among different applications. However, this approach has certain implications. By exclusively relying on spatial partitioning, the flexibility of FPGAs as a device becomes somewhat restricted since applications are fixed for the duration of deployment. Additionally, it assumes specific number of partitions and the amount of logic allocated to each partition, which may not always align with the dynamic requirements of applications. Moreover, spatial partitioning gives rise to other questions. For instance, when multiple applications share the FPGA, should they be allowed to communicate in a manner analogous to inter-process communication (IPC), and if so, what would be the mechanism for facilitating such communication?

Coyote Approach:

Coyote combines both approaches, providing a cooperative multitasking abstraction of a set of virtual FPGAs, each of which is timeshared between applications and services.

In this context, each "version" of the shell represents a specific configuration that includes a selected set of virtual FPGAs, their associated wrappers, chosen services, and the management layer. These shells are designed to be dynamically loaded or exchanged at any given time, and each is set up with a fixed number of vFPGAs (e.g. 1-8). These shells serve as the initial reconfigurable region within Coyote. Any reconfiguration of this region will implicate a reconfiguration of the application layer that operates within it.

Within these shells, the vFPGAs are organized as spatial partitions within the second hierarchical dynamic region of the chip. Each of these regions, for the purpose of executing user logic, is considered equivalent, much like cores in a symmetric multiprocessor system. Consequently,

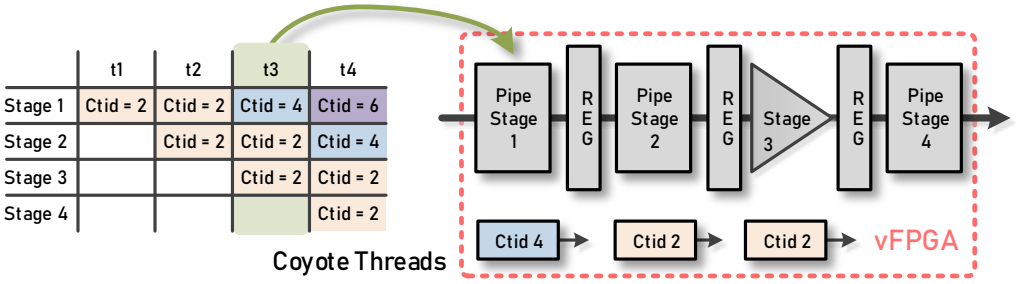

Figure 2.11: Hardware threads running in hardware process.

these regions are time-shared among different applications, allowing for efficient resource utilization and dynamic allocation.

Ideally, a single application bitstream could be loaded into any available "slot" for execution. However, in practice, a specific vFPGA bitstream is fully tied to a particular physical container, which is predefined within the chip's floorplan. While some research on relocatable bitstreams has been conducted [97] (although these experiments were performed on small SoC chips rather than larger Ultrascale, data center-oriented architectures), achieving true bitstream relocation is challenging given the current levels of FPGA heterogeneity. It's important to note that this feature is by no means officially supported in modern FPGAs.

As a result, at present, each application must be automatically synthesized in advance for each vFPGA slot, similar to compiling "fat binaries" for multiple architectures. We will delve into specific spatial and temporal scheduling considerations in the following sections, along with details about the execution environment provided for user logic.

We've established that vFPGAs are, to some extent, analogous to processes, but what about threads? How does the conventional concept of a thread, typically associated with a process, translate to the FPGA in the context of Coyote?

In Coyote, hardware threads function as distinct data streams that traverse through hardware pipelines. As a result, a single hardware process (vFPGA) can accommodate multiple individual hardware threads, all sharing the same user logic within the vFPGA. In this regard, they resemble OS threads, which share the memory space of the host process. However, it's important to note that, in context of memory, these Coyote threads offer more robust isolation properties compared

to their software counterparts. Different Coyote threads can run in various FPGA hardware and software processes, providing the memory isolation and enabling parallel execution. An example of Coyote threads running within a hardware process (vFPGA) is shown in Figure 2.11.

Discussion:

A scheme with this generality demands careful implementation. When timesharing vFPGAs, each dynamic region introduces some resource overheads because the boundaries between these regions typically require additional decoupling logic. However, beyond the resource considerations, the primary limitation here is the necessity for these vFPGAs to be fully floorplanned within the chip. This floorplanning process constrains the degree of freedom that the placer and router tools possess during the compilation flow. Designs with poor floorplanning can introduce considerable congestion and have a detrimental impact on achieving overall timing closure.

In the context of Coyote, floorplanning is predefined for all static layers (all hardware platforms), but the responsibility for determining how user applications should be floorplanned within dynamic layers is delegated to the developers who deploy the shells. This floorplanning process can be carried out either manually or with the aid of external floorplanning tools [78, 89]. However, it's important to acknowledge that the research in automatic floorplanning is still advancing, and it may not yet be fully equipped to effectively handle floorplanning for systems characterized by such a high degree of heterogeneity.

It is crucial to ensure that the overhead associated with context switching does not negate the performance advantages gained from employing a circuit initially. The process of dynamically reconfiguring an FPGA is relatively slow and is expected to remain so in the foreseeable future. We will quantify the cost of this overhead in the Section 3.3.3.

Moreover, it is essential that the logic required to enable multiple vFPGAs to communicate and share services in various chip regions does not consume an excessive amount of chip resources. Our current experience has been quite positive; we can efficiently run multiple useful applications on a single FPGA today. Furthermore, the hardware trends are in our favor, especially as FPGA components continue to grow in size [192]. For example, the hardware FPGA operators used for database acceleration in Farview [126] on average utilize less than 5% of the overall CLBs (Complex Logic Blocks) each. Without multiple tenants, substantial portions of the FPGA would remain unused.

2.3.2 Execution Environment

The process abstraction also serves the purpose of providing a *standard execution environment* for a program. This precisely defined execution environment streamlines the programming model and hides the complexity of the underlying system layers from end-users. A program compiled to run in a process can, in principle, execute in any process on any machine implementing the same execution environment. For example, in Unix, a process's execution environment consists of a virtual address space, one or more threads, a set of file descriptors, the system call interface, etc.

What's Different on an FPGA?

To date, there are almost no attempts to define a process-like execution environment for an FPGA. Most FPGA application development targets a specific model of FPGA. Porting the same logic to a different chip is often a non-trivial programming problem.

The heterogeneous nature of hybrid platforms complicates this question further. In addition to the environment in which user logic executes, a process abstraction must also address how software processes and FPGA-based logic "processes" interact across the hardware/software interface.

In GPUs, programming models like OpenCL and CUDA are the solution. Portability is raised to the compiler, and the execution environment is defined by the language in which the GPU code is written. This works well for GPUs because they function as pure accelerators. To some extent, this concept has been duplicated on FPGAs [197, 7]. However, it's important to acknowledge that replicating the restrictive GPU compute model on FPGAs is accompanied by numerous limitations.

Significant aspect that goes unnoticed in these frameworks is that hybrid FPGA-based systems aren't solely dedicated to computational tasks. For instance, they actively engage in I/O operations via network, memory, and storage interfaces – in fact, this capability to interface externally represents a key selling point. Rolling this functionality entirely into a compiler has not worked in conventional machines, and is unlikely to do so here. Instead, runtime interfaces are needed. Perhaps the closest GPU analogy here is ptasks [178], which present the GPU as a task-based runtime as opposed to a language-level OpenCL interpreter.

Coyote Approach:

For every hardware application (user kernels) deployed within vFPGAs, Coyote defines a single *Unified Logic Interface* (ULI), which serves as the hardware equivalent of an ABI (Application

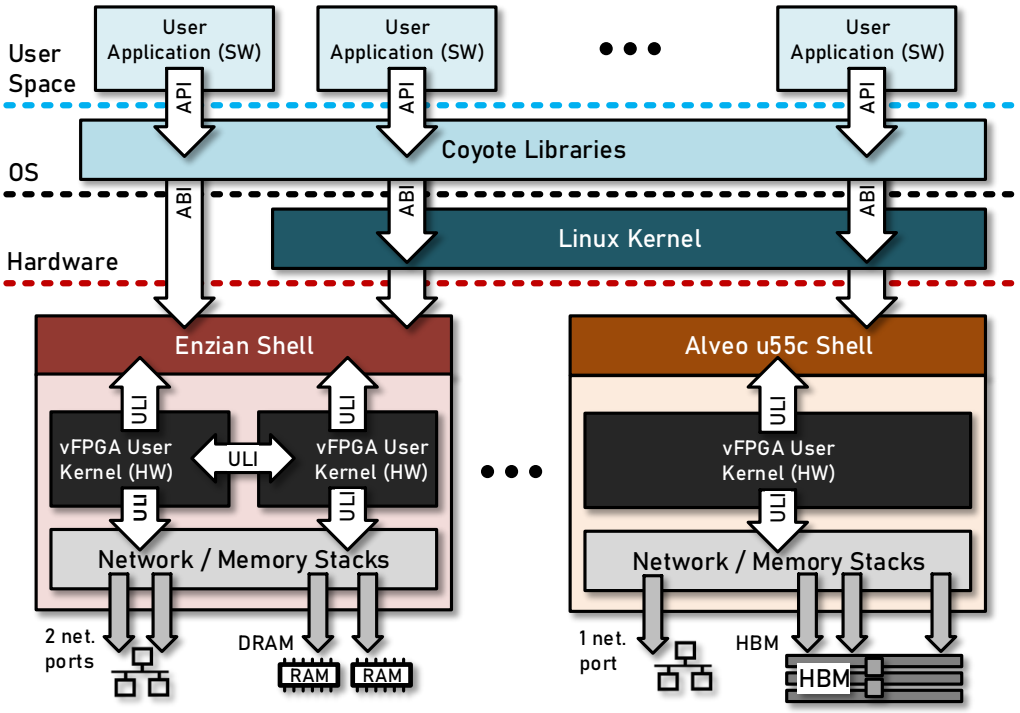

Figure 2.12: Unified Logic Interface within Coyote

Binary Interface). This interaction between this interface and the rest of the system is depicted in Figure 2.12. It employs the streaming AXI4 protocol, encompassing descriptors and data, to initiate bulk data transfers in all directions across the system, connecting with the host, memory stack, network stacks (such as TCP/IP and RDMA), and the user logic. Through this ULI, the user kernel establishes connections with the entire system through a collection of clearly defined and unified interfaces, effectively expanding the application's connectivity beyond the boundaries of the local FPGA node. This same interface is utilized for inter-region communication, facilitated by a control plane operating over an AXI4-light bus.

This interface is provided by the user wrapper in each vFPGA, and effectively sandboxes the user logic while providing communication with system services and memory – effectively combining functions of an address space and system call ABI in a software process.

Access to the ULI interface is exposed to application logic at a fairly low level, allowing read and write descriptors to be generated directly from the user logic in the FPGA fabric. Additionally, the accesses from the host software applications (including by high-bandwidth SIMD instructions) are also routed directly to the user logic.

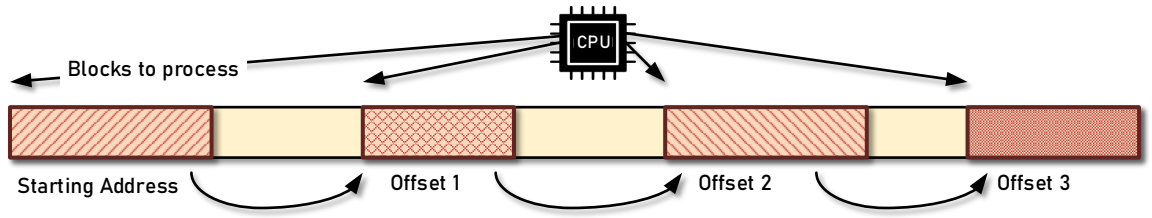

Figure 2.13: Example of pointer chasing: The CPU consistently interrupts the device at each hop to initiate necessary data transfers, leading to significant overhead and inefficient utilization of CPU cycles.

Crucially, through this ULI, user applications within vFPGAs possess the complete capability to orchestrate transactions throughout the system, including host memory, FPGA-side memories and the network. This is enabled by the virtualization layer in vFPGAs and the protective measures it offers. It provides a shared-memory model that greatly enhances the abilities of user applications for various processing tasks.

For instance, consider a scenario where an application is engaged in some form of pointer chasing (see Figure 2.13). In a conventional *host-centric system* [208], the CPU is responsible for managing every data transfer necessiated by the application. At each stage of data movement (each hop), the CPU must be either interrupted or has to poll in order to decide when to initiate the subsequent data transfer. This can introduce substantial performance overheads, as the communication with the CPU at each step can be quite cumbersome. Moreover, it results in a significant waste of CPU cycles, which is a particularly concerning issue, given that one of the primary objectives of using FPGAs is to offload operations and alleviate the workload on the CPU cores.

On the other hand, the host-centric model does have its uses, especially in sequential applications where bulk data movement is dominant. In such cases, it can be more convenient to rely on the host CPU for the control of data movement, as the associated costs in this scenario are minimal, while it eliminates the necessity for extra hardware and interfaces that are needed to supervise these data movement operations.

For this reason, in order to achieve maximum flexibility within Coyote, the ULI interfaces are extended to the host software API. This means that each software application has the capability to initiate calls using the same descriptors exposed in the hardware ULI. Consequently, Coyote gains the ability to initiate transfers from both software (following a host-centric approach) and hardware (utilizing a shared-memory approach). This stands as another distinctive feature of the

system, as the vast majority of systems typically allow only one of these approaches, either the former [208, 122] or the latter [146].

User software on the CPU interacts with the FPGA by creating a *task object*, essentially a closure consisting of user logic and other parameters and data. This is passed to the runtime manager for the execution on one of the vFPGAs.

Once functional, the user logic exposes a register interface in physical memory to the CPU, and the runtime manager maps this into the calling process' address space. Thereafter, the interraction between application software and user logic completely bypasses the kernel and runtime manager.

Discussion:

The ULI in Coyote incurs minimal overhead, but delivers considerable benefits, some of which might be surprising to those familiar with software development. It enables an approach analogous to microkernels, with common services provided to multiple vFPGAs over the AXI interconnect. Currently, Coyote offers support for memory (both DRAM and HBM) and network (TCP/IP and RDMA) stacks. However, the underlying concept here is that these services can be readily expanded to accommodate additional services using the same set of standardized interfaces that are already present in the Unified Logic Interface (ULI). We will examine these interfaces in greater detail in Section 5.1. Promising candidates for incorporating additional services include devices such as SSDs, NVMs, GPUs, and various other external accelerators.

As with conventional OSes, the Coyote execution environment also provides a way to deal with the evolution of hardware. The FPGA design space is changing rapidly. To take one example: in most FPGAs deployed in data centers today, the memory controllers and network stack (aside from PHY and MAC) are still instantiated as reconfigurable logic. However, both are becoming an almost universal requirement for cloud FPGA applications, which makes a strong case for building "hard" IP into future FPGAs to provide this functionality with less penalty in chip area - indeed, the latest design of Microsoft's Catapult platform offloads the full network stack to an ASIC (albeit off-chip and while sacrificing modularity). Intel's Embedded Multi-Die Interconnect Bridge (EMIB) extends FPGAs with dedicated machine learning accelerators [160]. Recent AMD Versal [32] cards also provide numerous off-chip hardware functions.

Identifying which portions of the stack are suitable for hardening while striking a delicate balance between maximizing performance and retaining the modularity inherent in reconfigurable hardware will be one of the major upcoming challenges in the future direction of FPGA development.

2.3.3 Preemptive or Non-Preemtive Multitasking?

Scheduling in conventional computing systems is a complex topic with a history older than computers themselves. In this context, our emphasis is directed towards the factors that influence scheduling mechanisms, as opposed to investigating into specific scheduling policies.

What's Different on an FPGA?

CPU scheduling can be preemptive or non-preemtive. Preemptive scheduling on CPUs requires a mechanism to interrupt a running process, save its state, and context switch to another, without any cooperation from the process or user program itself.

On an FPGA, such interrupt machanisms are not supported by any of the mainstream toolchains. Although in principle this possibility to read the state of the FPGA logic exists within hardware primitives like ICAP [13], it comes with massive latency and low level proprietary information which is difficult to parse and reason about. Some progress in this direction has been made in academia [125], but with significant performance penalties and implementation difficulties. Furthermore, the "state" of executing user logic potentially includes any stateful logic block (block RAM, URAM, flip-flops, DSPs, etc.) in the region of the FPGA used by the application, making the state capture all the more complex.

Efficient mechanisms with acceptable overhead to capture the arbitrary state of FPGA applications, ensuring they can be reliably resumed later, thus remain unclear.

Instead, existing approaches to timeshare an FPGA avoid preemption [196] and rely on two techniques. The first is a "task-based" approach where work units are submitted to the FPGA and run to completion much like Ptasks [178]. Secondly, as a last resort a badly-behaved piece of user logic can simply be deconfigured by the OS, in a manner analogous to killing a misbehaving process. The scheduling problem then becomes one of dispatching tasks to the FPGA. During this procedure, it's crucial to remove any residual state that may persist.

The key *quantitative* difference with FPGA scheduling is that context switch time is much higher: reconfiguring a dynamic region can take many milliseconds. Moreover, only one region of the FPGA can be reconfigured at a time. If not addressed, these limitations can lead to unacceptably high scheduling overhead.

Coyote Approach:

Coyote adopts the task-based technique, with tasks being described by *task objects*. Tasks are not scheduled by the FPGA itself, instead the runtime manager on the host CPU schedules them

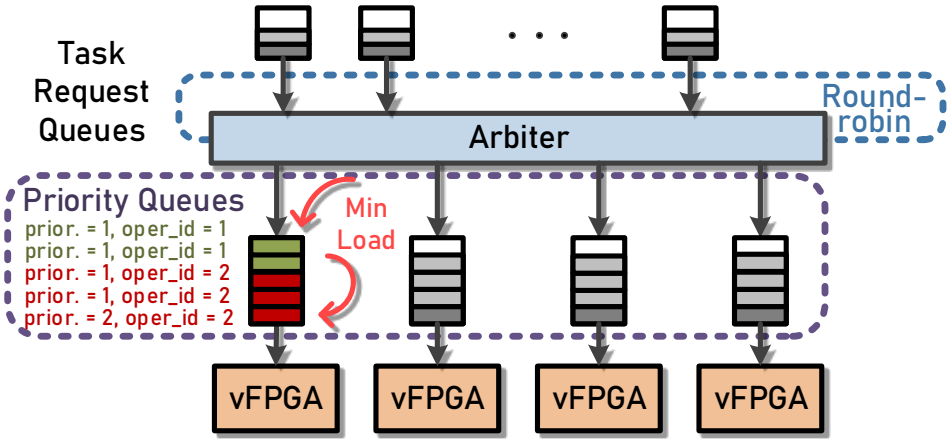

Figure 2.14: Modified priority based scheduling scheme employed in Coyote.

spatially (across vFPGAs) and temporally (by reconfiguring a vFPGA if required, and serializing such reconfigurations).

The current version of Coyote adopts a modified priority-based queue scheme for tasks (Figure 2.14). Application software submits a task to a per-application queue in the runtime monitor. These are serviced in a round-robin fashion, and dispatched to a priority queue for one of the fixed number of vFPGA instances. Each of these queues is sorted first by priority and, then, by the bitstream image that the task requires.

This heuristic provides a degree of fairness between applications (though it could certainly be improved with better protection against starvation in a few pathological cases), but more importantly groups together tasks that can run in a sequence without intervening reconfigurations of their vFPGA. This approximates some of the benefits of Optimus [146], which employs a more static assignment of logic to vFPGAs but shares this between applications. Note that it also makes the scheduler non-work-conserving.

Discussion:

Our scheduling approach notably reduces the number of unnecessary dynamic reconfigurations, as demonstrated in 3.3.3. Nevertheless, there remains important work to be undertaken to enhance fairness, prevent starvation, and enhance predictability in the scheduler. Unfortunately, dynamic reconfiguration has not yet reached the speeds required to effectively schedule short, burstable tasks, which we often find running in the OS. It does, however, find utility in longer-

running workloads that typically target accelerators. In such cases, the overhead associated with dynamic reconfiguration may have negligible impact.

Coyote deliberately avoids any question of preempting applications running in vFPGAs, except *in extremis* to "kill" badly behaving user logic. This decision is worth discussing in more detail, since other approaches [146, 122] provide explicit preemption interfaces. Applications can use these interfaces to implement user logic to save and restore their state in response to a preemption request from the scheduler.

The first reason for this decision is from an OS designer's perspective: the classical OS design principles adopted in Coyote strongly argue against this approach to preemption. Traditionally, user applications are not trusted to behave nicely by the OS, and so implementations take great care to ensure that preeemption never requires cooperation from the application – even in cases where it is explicitly visible to user threads, as in Psyche [148]. So-called "cooperative multi-tasking systems" (for example, early versions of the Apple Macintosh OS) do require application cooperation for context switching, but are generally not preemptive and, as history shows, are invariably supplanted by preemptive scheduling that does not require participation by the application.

The second reason is that the nature of "services" (e.g. networking) on an FPGA is different from that on a CPU. FPGAs emphasize spatial multiplexing and extreme concurrency. This means that services like the network stacks and physical memory management do not need to be scheduled in Coyote: they are separate circuits and so inherently run all the time. A user-supplied preemption implementation may appear sufficient where these OS facilities are absent, but their presence means that user-implemented preemption has to also save and restore state (such as network flows) in each of these services. This capture of system-wide state cannot yet be done efficiently in current FPGAs.

2.3.4 Virtual Memory

In a conventional OS, virtual memory provides a potentially unlimited number of "virtual address spaces" to software processes. By default, a virtual address space provides a sandbox of private memory, but segments of memory can be selectively shared between address spaces by the OS.

Virtual address spaces solve several crucial problems in computer systems: code and data does not need to be relocated at runtime, since it can be compiled and linked to run at a fixed ad-

dress. Demand paging to a disk or SSD allows the amount of memory seemingly available to all applications to exceed the total real memory in the system.

Fragmentation of physical memory is avoided at anything coarser than page granularity. Physical locations for data can be chosen carefully to provide cache-coloring transparently to user code. Accesses to memory regions can be tracked via a "protect-and-trap" technique, with applications ranging from garbage collection [37], copy-on-write, and transaction management [158] to dynamic binary translation [58].

Hardware support for the abstraction of virtual memory is traditionally provided by the MMU, by way of three key functions: *address translation* from a virtual to a physical address space, *protection* of memory pages, and a mechanism to *trap* to the OS on certain memory accesses (i.e. a page or protection fault).

What's Different on an FPGA?

Some uses of virtual memory do not make sense on an FPGA, such as trapping on particular instructions or memory addresses. However, others (demand paging, relocation, etc.) are highly relevant.

Existing approaches to programming FPGAs generally ignore virtual memory, or handle address translation solely in the host OS kernel [212, 86, 59, 64, 125, 162, 122, 210]. Pinned physical buffers are allocated and shared between FPGA user logic and software, which (when the data is not simply copied *en masse* between host memory and the FPGA) entails either the use of offsets to implement pointer-rich data structures, or cumbersome "pointer swizzling" when passing ownership of regions between devices. In both cases, one cannot simply pass a pointer from software to user logic without some mediation, typically by the OS kernel, or a runtime specific to a programming model like OpenCL [204].

One method for accessing host virtual memory from the FPGA is through the use of the host platform's IOMMU. However, IOMMUs are not well-suited for handling a dynamic set of FPGA applications, even those that exclusively utilize PCIe as an interconnect. Optimus [146] provides a comprehensive explanation of the limitations associated with IOMMUs and employs a technique called "page table slicing" to address these limitations. Other recent research endeavors also implement various forms of translation on the FPGA, enabling user logic to access host-physical addresses [199, 69, 167, 44].

These approaches, however, primarily cater to the scenario in which user logic accesses data residing in the host CPU's memory within a software virtual address space. Yet, modern FPGA platforms offer a wealth of additional memory resources closely integrated with the FPGA chip

itself, such as Enzian's FPGA, which boasts up to 1 TiB of DDR4 memory, as well as various devices like network or storage controllers.

The management of memory on the device side has been extensively explored in the context of GPUs [136, 65] and has reached a mature stage with the implementation of unified memory in CUDA [16]. This approach establishes a shared virtual memory space between the host and GPU (device side) memories, facilitating seamless exchange of user space pointers between them. Moreover, it abstracts the intricacies of hardware interaction away from end-users, eliminating the need for explicit data movement or custom memory allocation, greatly increasing the portability as well. To make this possible, support in the host OS kernel is essential, and Linux offers HMM (Heterogeneous Memory Management) as a solution [18]. HMM provides the necessary infrastructure to integrate device memory into the regular kernel path. While this approach has also been implemented for FPGAs in [116], it primarily tries to replicate the GPU compute model, without considering broader cases where, for instance, user logic is accessing host CPU memory. Additionally, it falls short in addressing concerns associated with multi-tenancy and networking. As previously mentioned, the attempt to mimic a GPU on an FPGA is proving to be a significant limitation, as it confines the FPGA to a subset of its features.

To ensure proper interaction with other OS abstractions, such as device virtualization, isolation, or straightforward access to FPGA resources from the CPU, a virtual memory abstraction for multi-tenant FPGAs must be extended to encompass all resources available on these heterogeneous platforms. It should support both hardware logic and software components while ensuring secure, consistent and reliable access to memory. This encompasses not only the FPGA's internal memory but also host memory, memory on the device side, devices connected to the FPGA, the network, and so on.

FPGAs also differ fundamentally from GPUs or other accelerators in that they are reconfigurable. In a CPU or, indeed, a conventional IOMMU or SMMU, parameters such as TLB (Translation Lookaside Buffer) size, associativity, coverage, etc. are fixed when the chip is laid out. They represent a careful, "one-size-fits-all" compromise intended to run most workloads reasonably well.

In contrast, with an FPGA TLB parameters can be changed on the fly to handle specific workloads more efficiently. Moreover, many accelerator workloads which deal with large data volumes benefit greatly from larger page sizes – this motivates Optimus [146] to use 2MiB pages exclusively, for example. These factors, combined with the lower clock speed at which FPGAs run (typically about 10% of CPU clock speed), make a *software loaded TLB* more attractive as a design option.

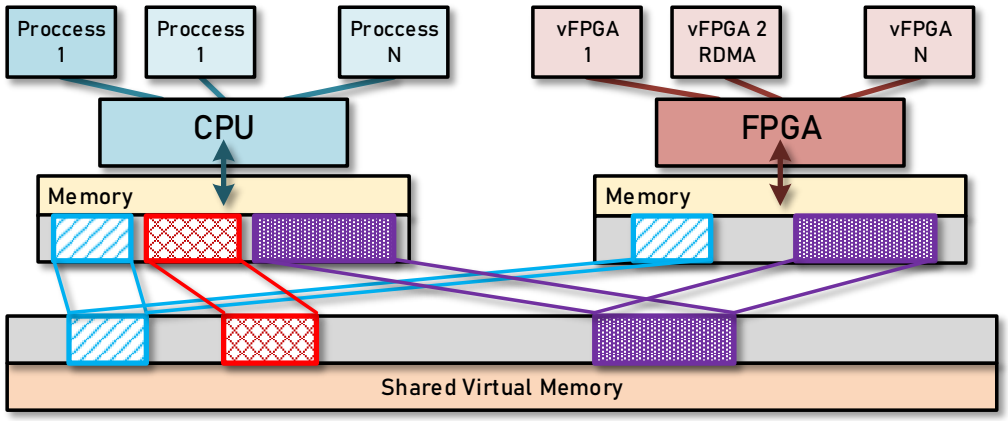

Figure 2.15: Shared virtual memory model used in Coyote.

Coyote Approach:

Rather than relying on the host CPU's IOMMU, Coyote implements a shared virtual memory model (Figure 2.15) that provides a dedicated MMU (Memory Management Unit) with software-loaded TLBs for each vFPGA. These TLBs are present in the wrapper of each vFPGA. This not only allows TLB dimensions to be decided based on the application, but also provides sandboxing of user logic regardless of whether it is accessing off-chip DRAM, HBM (High Bandwidth Memory) attached to the FPGA, or host CPU DRAM using the xDMA engines in the static region. It also makes floorplanning and routing easier on the chip by reducing fanout.

Moreover, the TLBs are positioned in the dynamic layer within the device so as to mediate *all* accesses to FPGA-attached devices and RAM, and the entire host CPU's physical address space, something not possible with a conventional IOMMU.

The high-level operation of TLBs in Coyote is best described in two parts: first, the underlying mechanism, and second, the different memory usage models it supports.

Mechanism:

The TLBs are implemented using on-chip block RAM, ensuring efficient memory access for any access pattern. They support multiple page sizes providing a broad coverage of memory addresses, especially crucial with limited on-chip memory resources. The associativity and number of sets are determined within the dynamic layer and can be resized during runtime.

When a TLB miss occurs, it triggers an interrupt to the host CPU. Subsequently, the driver identifies the specific vFPGA responsible for the fault, and it either populates the TLB with a

valid mapping or signals a page fault. In the event of a page fault, the host's software stack takes over further necessary steps to resolve it.

All accesses to both FPGA and host DRAM *from user logic* use the same unified TLB interface. User logic can therefore access any host memory, if the TLB allows it. Accesses to both host and FPGA memory are routed along separate paths to enable concurrent operation, which is crucial for fully utilizing the available bandwidth.

Meanwhile, on the *host* side, the CPU's physical address space contains a region for each vFPGA, each of which is further subdivided into three parts:

1. The TLB contents and other privileged configuration values. This subregion may only be mapped by the privileged Coyote device driver.
2. Registers within the dynamic layer and user wrapper accessible to user software, e.g. for setting up DMA copies.
3. Direct access to user logic. CPU-initiated accesses are presented to user logic as AXI4 transactions to be interpreted as the user logic sees fit.

Usage models: The most common way of using this facility in Coyote is to provide GPU-style "Unified Memory", but with FPGA flavour. This is essentially a form of local distributed shared virtual memory: pages are copied (faulted in) on demand between FPGA and host memory via DMA with coherence managed by a combination of driver software and dedicated "page fault" units in the secure wrappers.

Importantly the memory model in Coyote differs from GPUs as it additionally alllows direct "streaming" access to the host memory without necessating a copy to the FPGA memory. TLB entries thus indicate that, when a corresponding virtual address is requested, the physical access is directly routed to either host or FPGA memory without any extra copying. For efficient random access (such as pointer-chasing) this may be much faster to program and execute than "unified memory".

Coyote can thus handle multiple application contexts maintaining different shared virtual address spaces. It is the job of the software component within Coyote to ensure that address mappings are consistent between the vFPGA TLBs and the virtual address space of the corresponding software processes. This allows direct sharing of pointer-rich data structures between application software and user logic in hardware.

Finally, it is also possible to route fine-grained CPU accesses to addresses on the FPGA back through the vFPGA wrapper's TLBs and into FPGA (or, indeed, host) memory. While quite slow on PCIe-based systems, it might be an attractive option on a fully-coherent non-PCIe system like Enzian.

More in-depth view at Coyote's shared-memory model will be covered in Section 4.2.

Discussion:

The TLBs impose very little space overhead in FPGAs, and deliver in return considerable simplicity in programming applications. The partitioning of TLB functionality across vFPGAs brings a degree of performance isolation to vFPGA applications: one vFPGA cannot pollute the TLB contents of another region and thereby impact performance, an important consideration for a multi-tenant environment.

Note also that the area occupied by TLBs can be traded off against performance in an application-specific manner. This would not be possible with conventional IOMMUs situated on the PCIe bus, and would be hard to achieve with a single IOMMU shared between applications. Partly as a consequence, we have yet to see serious performance overheads due to software-loaded TLBs.

2.3.5 Memory Management

In addition to virtual memory, a traditional OS provides facilities for managing *physical* memory. As hardware has become more complex, this has become more important for performance. E.g. an application might request regions of contiguous RAM to optimize sequential access and/or TLB coverage via superpages, or memory on specific NUMA nodes, explicitly (e.g. via `libnuma` on Linux) or implicitly (e.g. using Linux' "first-touch" allocation policy).

These abstractions more concern performance than correctness. However, in the case of peripherals and heterogeneous accelerators it may be a requirement for software to work. A CUDA application may need to allocate GDDR memory on a GPU which is accessible (over PCIe) to CPU code. Alternatively, a device might only be able to DMA to and from a subset of the CPU-attached physical RAM.

In a software OS, however, there is a *single* mechanism available to allocate memory with the right characteristics: choosing an appropriate range of physical addresses. The physical address functions as a proxy for all kinds of features of the hardware interconnect, memory controllers, DMA capabilities, and (in the case of cache coloring) the processor's cache architecture and placement policies.

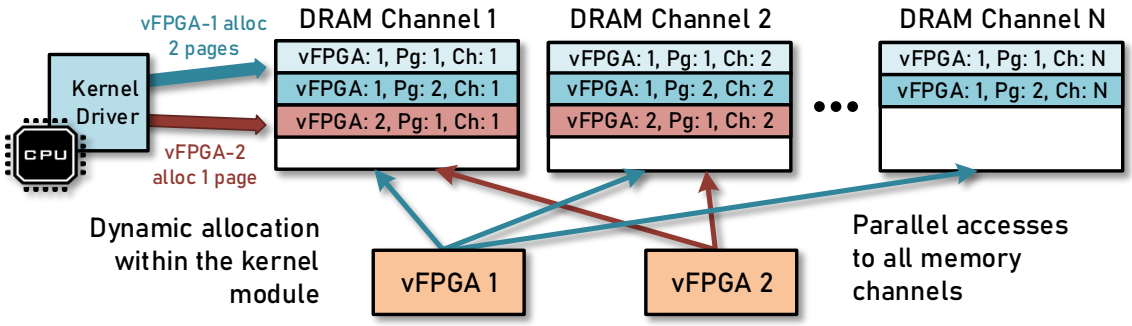

Figure 2.16: Dynamic allocation and parallel access across available channels in Coyote.

What's Different on an FPGA?

The situation differs significantly when it comes to FPGAs since they are much closer to the hardware. In the context of FPGAs, code running on them has the capability to access memory controllers directly. Unlike traditional systems where data paths are constrained by cache lines, machine words, or MMU pages, FPGAs offer a higher degree of flexibility.

For instance, SDAccel [7] intentionally exposes memory controllers to programmers, granting them greater control and flexibility. However, this approach sacrifices simplicity and portability across devices. Additionally, the memory accessible to programmers via this method is typically addressed using physical addresses and often lacks built-in mechanisms for ayn kind of sharing or protection.

The memory potentially visible to FPGA user logic is much more diverse than in software (and there are no caches). Block RAM (BRAM) is fast but scarce, DRAM is slower but there is typically much more of it, many systems have extensive off-chip DRAM available, and newer FPGAs incorporate high-bandwidth memory as well.

Moreover, as with servicing TLB misses, it may be useful to offload the dynamic allocation of FPGA memory to software, although some hardware allocators exist [209].

Coyote Approach:

Allocation of physical memory both within and between vFPGAs in Coyote is handled by software, in the kernel driver, which also takes care of creating virtual memory mappings both for the user logic and application CPU code. A variety of physical memory types can be used (off-chip DRAM, host DRAM, HBM, etc.).

Accessing memory is similarly different. Whereas software deals with register loads and stores

or cacheline fills and writebacks, *any* memory access in FPGA user logic is inherently, and explicitly, a copy operation from one location to another.

On current PCIe-based systems, user logic can access the entire host CPU's physical address space (albeit subject to memory protection) by transparently using xDMA copy engines.

The complex task of routing memory accesses originating from both user logic and host software, whether directed towards host memory or FPGA resources, is efficiently managed by a dedicated *read/write engine* running on top of each vFPGA wrapper. This engine is responsible for providing the necessary flexibility in memory access. Requests are submitted to the read/write engine using base/length descriptors. Accesses from host software to FPGA memory are translated into these descriptors by the interface logic, whereas user logic can issue them directly. This approach minimizes overhead by keeping operations within the realm of virtual addresses, which are essentially user space pointers. This design choice greatly simplifies the development of Coyote kernels.

On fully coherent systems like Enzian, the read/write engines would be replaced with the interface Coyote provides to the CPU's native cache-coherence protocol.

In contrast to approaches like SDAccel (but more in line with a software environment), Coyote hides the presence of individual on-board DRAM controllers from user logic. On-board DRAM is the most commonly used way to hold bulk data in most FPGA acceleration algorithms, since it is higher capacity than BRAM. Coyote aims simply to maximize the bandwidth of bulk sequential access to this resource for user logic running in a vFPGA.

Coyote *stripes* DRAM access across all available controllers via careful allocation of pages. Each page is then striped across channels – e.g. if the FPGA has two physical DRAM channels, the first 1MiB of each page will access one DRAM channel, and the second half will use the second channel. This permits bandwidth optimization when performing rapid accesses with multiple channels present, and results in considerable performance gains over the naive approach. Accesses from different vFPGAs are still interleaved at each memory controller, as shown in Figure 2.16.

Crucially, the type of memory attached, whether it's standard UDIMMs, RDIMMs, LRDIMMs, or even high-bandwidth memory from newer devices, doesn't affect the user-facing interface provided by Coyote. It remains consistent and unchanged regardless of the memory type used.

Discussion:

User logic is rarely as "memory-allocation intensive" as software, and so the kernel memory allocation code is rarely on the critical path.

By abstracting away on-chip DRAM controllers, Coyote makes a tradeoff in favour of portability and ease of programming, which we argue (based on our experiments) is appropriate. Moreover, Coyote applications can directly run on future FPGA designs which entirely offload memory controllers to dedicated hardware.

In this context, striping provides more than just faster sequential access: it is vital for abstracting and sharing memory controllers since it allows the DRAM controllers to enforce fair sharing of bandwidth between vFPGAs.

2.3.6 Network Stacks

The main advantage of operating system abstractions lies in the services they offer, enabling programs to be written without needing to deal with the intricate details of underlying hardware. In today's cloud-dominated landscape, networking services stand out as one of the most crucial.

Although the network stack is not an integral part of the Linux kernel, the TCP/IP stack within it is a critical component. It is challenging to imagine a real world deployment of the kernel with any meaningful applications running on top of it, without this stack. As networking becomes increasingly essential with the ever-expanding volume of data, the performance of this stack becomes a de facto bottleneck for many applications [112].

As a result, various methods have been explored to enhance its performance, as the kernel-related overheads can be substantial. One of the most prominent approaches is the "bypass" method, which has been employed to circumvent these overheads. Intel's DPDK (Data Plane Development Kit) serves as an exemplary illustration of this approach [11]. In an effort to reduce overhead and boost performance, DPDK shifts large portions of the network stack from within the kernel to user space. Additionally, it adopts the polling mode to eliminate the expense of interrupts.

Despite the improvements achieved by this approach in speeding up the traditional TCP/IP stack, it still, ultimately, places a burden on the host CPU for packet processing. Wasting CPU cycles is a significant concern in modern data centers, as highlighted in [117], where up to 30% of the CPU resources are wasted on system tasks that are otherwise prime candidates for hardware acceleration. In this case, network stacks are no exceptions.

What's Different on an FPGA?

FPGA devices, especially within heterogeneous systems, present an ideal opportunity to offload the entire network stack, thus relieving the CPU from the burden of network management. The

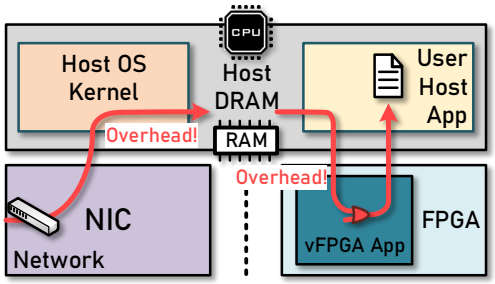

Figure 2.17: FPGA as a conventional accelerator for in-network processing.

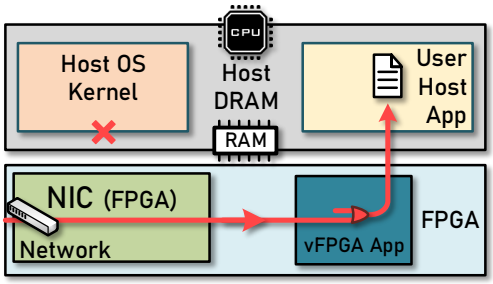

Figure 2.18: FPGA directly attached to the network, used as a smart NIC.

deep pipelines generated for packet processing within network stacks align well with the processing capabilities of FPGAs. Importantly, these FPGAs lie on the path of the data, they are connected both to the network and the host CPU memory, allowing them to process data as it flows through the device.

The overhead introduced by this FPGA-based processing is minimal, primarily because the latency is largely concealed by the overarching network and memory operations. This ”bump-in-the-wire” type of network processing is an area where FPGAs shine the most. Their flexibility is particularly valuable, as complex network stacks often require additions and modifications that could make an ASIC implementation prohibitively expensive.

Furthermore, an additional advantage is that extra processing can be easily incorporated atop these stacks as data moves from/toward the CPU, and this additional processing can be performed cost-effectively using FPGAs.

RDMA (Remote Direct Memory Access) introduces a higher level of abstraction compared to the traditional TCP/IP stack. RDMA is a technology that facilitates communication between user processes on remote machines without relying on the local CPUs. In other words, it enables two processes to exchange data stored in user space buffers without any CPU intervention. In contrast to TCP/IP, RDMA completely bypasses the kernel, resulting in minimal overheads, while importantly providing a very high level of abstraction at the user level.

RDMA aligns well with Coyote as it offers communication at the user space virtual address level, similar to the communication between vFPGAs and the host CPU already present in Coyote. This alignment makes RDMA a well-suited networking abstraction, enabling efficient and high-performance communication, with added potential for FPGA side processing, while preserving high level of abstraction exposed to the end users.

Coyote Approach:

The offload of networking stacks has been explored within our group to a great extent resulting in the development of a fully offloaded TCP/IP stack [183, 184] and RDMA stack [187]. Both of these stacks are implemented using HLS and are fully realized in the FPGA fabric. These stacks leverage high-performance 100 Gbps transceivers that are commonly available on most modern FPGAs, including the complete range of Alveo cards, as well as our own Enzian boards. These speeds more than meet the requirements of modern data center environments [190].

The integration of these stacks into Coyote was a natural fit, given that the platforms for which these stacks were originally developed are the same platforms targeted by Coyote. The primary challenge lay in extending these stacks to operate effectively within a multi-tenant environment while maintaining the high-performance characteristics they offer. Moreover, as Coyote was specifically designed for data-center and cloud environments, ensuring the reliability of these stacks within such expansive and complex environments was imperative. Detailed examinations of the architectures of these stacks will be provided in Chapter 4.

Discussion:

While modern FPGAs offer high-speed, low-level network interfaces ideal for networking operations, the availability of higher-level networking abstractions is severely limited. This limitation often arises from the mindset prevalent in FPGA development, which tends to focus on low-level details, often overlooking the potential benefits of higher-level abstractions.

The advantages of these network stacks are evident. Beyond the significant benefit of alleviating CPU load, which is one of the primary advantages of offloading networking tasks to FPGAs, the second major advantage, particularly when compared to commercial Network Interface Cards (NICs), is the ability to incorporate additional custom processing on top of these stacks. Importantly, this processing is not constrained by any fixed, hardened cores within the FPGA (as is the case in commercial NICs) and can be tailored to suit the specific requirements of arbitrary algorithms. This flexibility represents a substantial advantage of FPGAs over ASICs. Furthermore, FPGAs themselves do not pose bottlenecks either. Modern FPGA speeds, which can even reach up to 400 MHz with very wide channels, are more than sufficient to effortlessly match the speeds of modern interconnects, memory, or networks.

2.3.7 Inter-Process Communication and Additional Services

A traditional OS provides a number of abstractions beyond those we have covered here. Among the most fundamental ones, at the heart even of microkernel architectures, are Inter-Process Communication (IPC) and Input/Output operations (I/O).

Coyote Approach:

We have already described how Coyote provides communication between vFPGAs and CPU-based software processes, but it also allows optional hardware queues between vFPGAs by analogy with IPC channels, pipes, etc., in a manner reminiscent of Centaur [167]. This allows users e.g. to chain dataflow operators running in different vFPGAs together while preserving the isolation between them.

While inter-vFPGA queues can be used for inter-application communication, we find they are rarely used as such. This is primarily because of the increased routing complexity that arises when additional connections need to be established between vFPGAs. This complexity significantly complicates their floorplanning and overall system architecture.

An alternative approach is to utilize shared virtual memory on the FPGA side. Since FPGA side memory is virtualized in Coyote, custom allocations can be done, enabling the sharing of pages between vFPGAs and facilitating efficient inter-process communication without introducing routing complexity or consuming additional resources.

Discussion:

As with containers, in our experience inter-vFPGA communication, when it happens at all, is coarse-grained and benefits from being independent of whether the vFPGAs share the same FPGA. It is important to note that this feature is optional and, unlike in the traditional OS, is something that is seldomly used.

Instead, the main use for such queues is communication with *services* provided by Coyote. For instance, both the memory and network stacks abstract the intricacies of the underlying physical interfaces and present a standardized, portable interface that is shared among all the vFPGAs in the system. This shared set of services serves as the point of interaction for these vFPGAs and can be facilitated for inter-process communication.

Imporantly, further services can be similarly implemented and configured into the dynamic layer at startup, for example a storage stack (perhaps driving directly attached Flash memory).

These additional services do not necessarily have to be integrated directly within the FPGA itself. A notable illustration of this concept is the interaction with modern GPUs, which provide

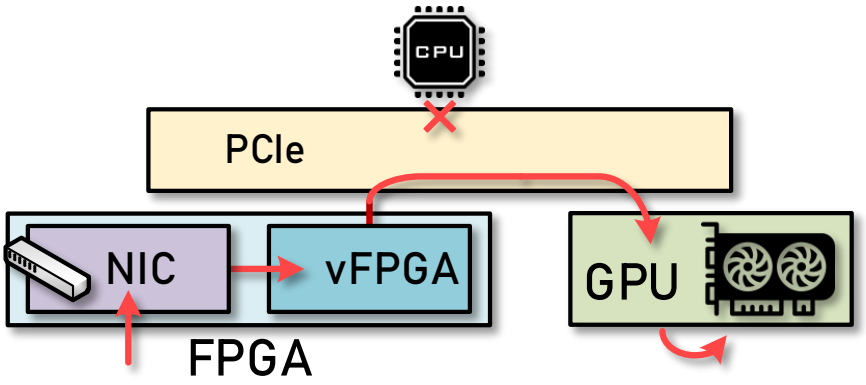

Figure 2.19: Direct FPGA-GPU communication.

technologies like GPU RDMA direct [17]. This technology facilitates direct communication between the GPU's internal DRAM and an external DMA capable device without requiring CPU intervention. This feature is especially compelling for Coyote, as it allows for the incorporation of additional preprocessing on top of RDMA. This preprocessing can occur within the FPGA before the data reaches the GPU. Importantly, Coyote's architecture fits well to these operations as the dedicated MMU layer ensures secure access to these external devices, provided that these devices expose their internal memory, which is the case with newer GPUs such as Quadro and Tesla range of devices from Nvidia [161]. This capability opens up numerous intriguing processing possibilities, all without incurring additional data movement, as depicted in Figure 2.19.

Importantly, incorporating such capabilities into Coyote doesn't require extensive changes, as the virtualization layer inherently supports this concept. With the addition of the appropriate mappings to the TLBs, the rest of the system can seamlessly accommodate these functionalities.

The microkernel analogy applies here: Coyote provides a basic framework where such services can be added in the future based on use-case requirements.

Unlike in a software-based OS, Coyote "services" like the network stacks do not need to be scheduled, since they are always present on the FPGA – the FPGA is being spatially rather than temporally shared between services and user logic.

2.4 Related Work

The FPGA community has generated a tremendous amount of work in recent years on programming and managing FPGAs. We have compared Coyote with many examples already in Section 2.3, and two recent surveys [115, 196] give an excellent overview. In this section, therefore, we focus on a few important recent systems.

The initial version of Microsoft's Catapult[172, 62] environment offers a reusable, static portion of programmable logic accessible through a high level API. Configurability for the (single) application is possible for modules like the network (recently offloaded to a sophisticated ASIC) and memory. The accelerator/smartNIC usage model means there is no support for virtualization nor partial reconfiguration.

Intel's hybrid CPU-FPGA HARP design [162] turns the FPGA into one more processor. Intel implements its own QuickPath interconnect [110] supporting full cache-coherent memory access, but only to external memory on the CPU side. Usage model and management is similar to Catapult. Partial reconfiguration is possible but there is no option to include local on-board memory or network modules on the FPGA.

Xilinx SDAccel [7], used by Amazon [30] and Alibaba [26] in their cloud deployments, also divides the FPGA into a static "shell" and dynamic user regions. One application can run at a time, but the user logic can be exchanged at run time with the help of partial reconfiguration. To date, there is no support for I/O devices or network.

All these deployed systems support only a single application at a time, and also do not try to provide a shared virtual address space between host software and user logic. Systems in the research literature are rather more ambitious in adopting one or more ideas from traditional operating systems:

AmorphOS [122] aims to increase FPGA utilization by placing multiple applications on the FPGA. It provides protection on FPGA-attached memory, but no access to host memory. Protection is based on segments set up by the host OS. AmorphOS can operate in "low-latency mode", where applications occupy different parts of the dynamic region, and "high-throughput" mode, where everything is synthesized into a single bitstream. Time-division multiplexing in low-latency mode is achieved by requiring applications to implement correct checkpoint and resume.

AmorphOS can be seen as pushing many traditional OS problems into the synthesis pipeline, and compiling many different bitstreams for configurations which, in Coyote, are handled at

runtime by the same image. Since it provides no integration with the host memory system, and applications are directly compiled to the FPGA, AmorphOS provides no virtual memory facilities beyond segmented addressing of FPGA memory. Scheduling is simplified by not partially reconfiguring the FPGA, which also obviates the need to provide a uniform network interface.

AmorphOS optimizes how many applications can fit on one FPGA, at the cost of compilation and deployment overheads, by delegating OS functionality to synthesis tools. In contrast, Coyote's OS-centric approach standardizes the execution environment, allowing applications to be flexibly deployed, and evaluates the cost of this generality.

Optimus [146] provides FPGA user logic with access to host memory via a per-application virtual address space. It partitions the dynamic region into application containers which appear not to be partially reconfigurable, but which can run the same user logic on behalf of multiple applications. As in AmorphOS, user logic implements checkpoint and restore to allow time-division multiplexing of resources. Optimus allows the host address space to be shared, but does not give a host process access to the address space of a vFPGA.

Optimus has many similarities with Coyote, but focuses on a subset of OS functionality. By avoiding dynamically reconfiguring vFPGAs, scheduling is simplified relative to Coyote. Optimus provides address translation, but only for vFPGA-to-host access, whereas Coyote provides a true unified virtual address space shared between host process and user logic. This in turn allows Coyote to virtualize services like the network stack, something Optimus does not do. Optimus therefore does not provide a standard execution environment for bitstreams, since the functionality it does provide would not benefit from such an environment.

ViTAL [210] focusses on clusters of FPGAs and, unlike Coyote, addresses distributing applications across a cluster. While it provides a network device, and flexible multiplexing, it does not target hybrid CPU/FPGA systems, and provides neither unified memory, nor (e.g.) a shared network *stack* between vFPGAs.

ViTAL virtualizes access to the FPGA memory and the network *device* (as a simple point-to-point communication link). As with Coyote, it uses a fixed partition of the dynamic region in reusable vFPGA which are allocated to applications when they are deployed. A key feature of ViTAL is being able to partition applications and compile them into multiple vFPGAs; these can then be deployed on the same FPGA or several connected by point-to-point links.

By not supporting host memory access nor virtualizing a high-level service like TCP or RDMA, ViTAL is relieved of the need for a virtual memory system. Moreover, by using the compiler to turning a set of physical FPGAs into one large logical FPGA by application partitioning, it

obviates the need for a standard execution environment.

To greater or lesser degrees, all these systems focus on optimizing one or another metric and implement a subset of the critical functionality of a classical OS.

In contrast, Coyote investigates the consequences of a complete, general-purpose approach: putting a general OS feature set together (multi-tenancy, multi-user TCP/IP and RDMA stacks, unified memory translation/protection across CPU and FPGA, dynamically allocated memory stack, standardized execution environment, etc.). Uniquely, this combination is what allows Coyote to provide shared high-level OS services like networking.

It also demonstrates that a full set of combined OS features fundamentally changes how a system like Coyote is designed, and this is where it differs most from prior work while still reusing a number of ideas from such systems.

2.5 Summary

In this chapter, we introduced various conventional abstractions and examined their efficient application in the context of modern heterogeneous systems. We showed the consequences of lacking these abstractions by highlighting several key limitations found in modern cloud deployments that incorporate FPGAs.

To address these limitations, we have created Coyote, a *shell* aimed at establishing a comprehensive "microkernel"-like environment for these devices. Coyote is created to streamline the utilization of these systems and offer a comprehensive multitasking setting to fully exploit their performance. In the upcoming chapter, we will begin our exploration of the practical implementation of the aforementioned abstractions in Coyote.

Chapter 3

Static Layer: The System's Kernel

In this chapter, we will explore the "static layer" of the system. This foundational layer interfaces with platform-specific hardware, facilitating the functioning of the higher levels of the stack. Additionally, the host device driver is an integral component of this layer, serving as the primary interface between software and hardware applications. In many respects, this layer is a core component of Coyote, much like the kernel serves as the core of an operating system.

The static layer encompasses vital components required for the subsequent loading of FPGA images and user-defined shells. Particularly noteworthy are the CPU-FPGA interconnect and reconfiguration controllers, which play a pivotal role in this context. The initial Coyote hardware image, loaded onto the device, consists of the static layer in conjunction with a "dummy"

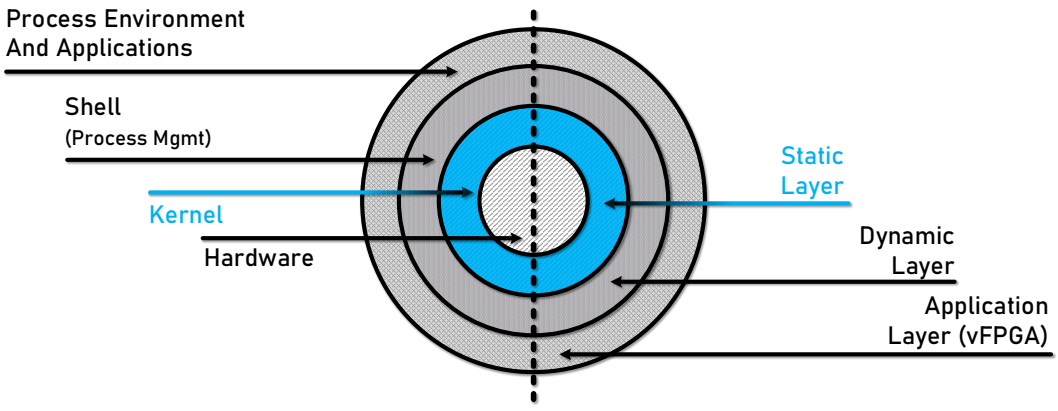

Figure 3.1: Static Layer.

dynamic layer. Subsequently, users have the responsibility to replace the initial dynamic layer with user-deployed shells containing their own applications in later stages.

We will start this chapter by focusing on the CPU-FPGA communication. This link serves as the primary channel connecting the host CPU to the circuits deployed in the FPGA. Its role is pivotal in managing most operational tasks and facilitating the essential data exchange between the host memory and the accelerators. An important aspect of this link is the infrastructure constructed on top of it, as well as the interfaces that are made accessible to the "higher" dynamic layer.

We will then demonstrate portability of Coyote's interfaces across a variety of platforms, not limited to a single vendor's offerings (range of AMD data center cards). To this end, We are also evaluating its compatibility with a custom, in-house-developed heterogeneous research computer called Enzian [70], featuring a novel interconnect and a diverse range of unique capabilities. We will delve into Enzian's offerings, explore how Coyote was adapted to work with this platform, and conduct performance comparisons between different platforms. This will also demonstrate the potential for expanding Coyote's compatibility to other vendors and platforms, such as Intel, as well as emerging cache coherent interconnects.

Lastly, Coyote's unique hierarchical dynamic reconfiguration will be examined. It provides Coyote with the ability to swap shells during runtime while maintaining dynamic application reconfigurability. This hierarchical approach enables us to independently provision active circuits and efficiently segregate system services from user applications. In essence, it provides us with a means to construct a hierarchical stack with multiple layers within the FPGA.

3.1 Communication with the Host: The FPGA's "Lifeline"

One of the primary advantages of heterogeneous systems lies in their ability to adapt to diverse workloads. These workloads can be inherently varied and may execute different portions on separate devices, such as both the CPU and FPGA. Consequently, the communication link connecting these devices becomes a vital component in heterogeneous systems, as it handles a substantial portion of data movement. Moreover, while FPGA fabric offers flexibility and high computational performance, it may not be the most convenient choice for management operations. In this context, the host CPU plays another important role in a hybrid CPU-FPGA system. It effectively manages a wide array of administrative tasks and, through its device driver and interfaces to the FPGA side, controls the overall operation of the entire system. This complete orchestration is typically achieved through a high-speed communication interconnect.

In most modern cloud environments, the CPU-FPGA interface is established using PCIe (Peripheral Component Interconnect Express). This high-speed serial bus standard is employed to interconnect a wide array of hardware components. One of the notable advantages of PCIe is the support for various devices, making it a common choice for connecting graphics cards, storage devices, network cards, and, of course, FPGAs.

PCIe relies on a serial communication method, where data is transmitted one bit at a time across multiple lanes. This approach results in significantly enhanced data transfer rates when compared to older bus standards. PCIe is further distinguished by its robust error-handling mechanisms, encompassing both error detection and correction, which ensure the reliability of data transmission. Notably, PCIe offers full backward compatibility, enabling newer devices to function seamlessly with older standards. PCIe is also well-supported by most modern operating systems, and custom driver development can take advantage of the rich existing API, making it easier for hardware manufacturers to create drivers for their devices.

This is in sharp contrast to emerging interconnect technologies, which may lack this level of resilience. PCIe's robust and adaptable features have made it a trusted and widely adopted communication standard. In summary, PCIe is a versatile and high-performance hardware protocol that has become a standard for connecting a wide range of hardware components in modern computers. Its features like scalability, hot-plugging, and compatibility make it a good choice for interconnecting various hardware devices within a computer system.

Thanks to this versatility, the PCIe protocol is widely utilized in modern FPGAs. Typically, these connections utilize multiple lanes, each of which is bidirectional (one to transmit and one to receive). The flexibility of these lanes allows for various configurations to accommodate different hardware components, with the number of lanes determining the overall connection bandwidth. FPGAs, known for their flexibility, can easily support most of these configurations.

It's important to note that all Alveo cards support Generation 3 PCIe with 16 lanes, providing approximately up to 16 GB/s of theoretical bandwidth. While not the latest standard, these speeds are more than sufficient for handling modern data center workloads and are in line with the high-speed network bandwidths commonly found in today's environments.

Modern FPGAs often integrate a hardened high-performance PCIe core that exposes the PCIe interface to the FPGA fabric without utilizing any of the fabric's resources. In the case of Alveo boards [12], which Coyote is specifically designed to work with, the supported core is the PCIe integrated block for UltraScale+ devices [33]. This core not only complies with the modern PCIe standard but also offers extensive customization options to align with specific user requirements.

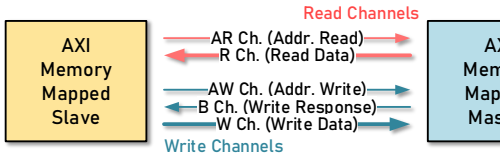

Figure 3.2: AXI memory-mapped channel.

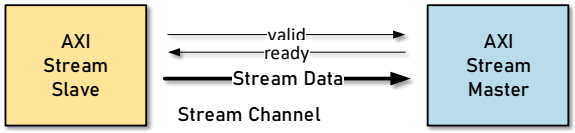

Figure 3.3: AXI stream channel.

3.1.1 Interface: Memory-Mapped or Streaming?

Once PCIe has been established as the communication method, the subsequent question is: "What type of interface should be constructed on top of it?" One of Coyote's core goals is to ensure its versatility across diverse platforms, without any reliance on particular hardware device or type of interconnect. Consequently, it necessitates the development of a higher-level interface that goes beyond exclusive dependence on PCIe.

For this reason, the interfaces exposed by the Coyote shell to the platform are primarily based on derivatives of the *AXI protocol*. This choice is primarily driven by the widespread adoption and strong support that the AXI protocol enjoys within the hardware domain. The protocol offers several advantages, including efficient pipelining that minimizes potential bubbles or idle cycles, making it particularly well-suited for optimizing data throughput in both memory and bus transactions.

The protocol essentially depends on a straightforward "handshaking" logic. Data is propagated only when both the slave-driven valid signal and the master-driven ready signals are asserted. If the ready signal from the master is not activated, it creates backpressure toward the slave. This is a way to efficiently manage and control the flow of data within the system, preventing data from being transferred faster than the receiver can handle. Backpressure ensures that the data transfer remains efficient and doesn't overwhelm the system, particularly when there are disparities in processing or data handling capabilities between different components. It allows for coordination and synchronization of data flow in a way that maintains system stability and prevents data loss or errors. This is especially important in Coyote, where various system components can operate at different clock speeds and different processing rates.

The intrinsic value of the AXI protocol lies in its remarkable flexibility. For instance, it offers the ability to select different types of accompanying signals, which can aid in stream identification, routing, caching information, burst types, etc. This adaptability allows the protocol to be

finely tuned to align with the specific demands of the system in question. Significantly, the AXI protocol offers a range of variations, supporting both memory-mapped interfaces, as depicted in Figure 3.2, and streaming interfaces, as shown in Figure 3.3.

The "Memory-Mapped" and "Streaming" interfaces, both derived from the AXI protocol, represent distinct flavors of the protocol utilized for data transfers in computer systems. The comparison of these two interfaces is outlined below.

Memory-Mapped:

- This interface maps registers and memory locations within the FPGA into the CPU's memory address space. As a result, they become accessible over the interconnect, just like any standard memory locations within the CPU domain. Suitable for any configuration plane.
- It incorporates five channels: two address channels designated for read and write operations, separate read and write data channels, and a write response channel used to indicate the completion of write operations. While these additional channels are necessary for the memory mapping interface, they can notably increase the number of extra bits and signals which get routed through the FPGA. This can also lead to an increase of the logic employed to trigger transfers, potentially introducing some overheads and latency, especially if relying on the CPU initiation.
- On the other hand, the memory-mapped interface provides fine-grained access and is self-sufficient, eliminating the necessity for resource-intensive DMA. This attribute can be particularly advantageous for fine-grained workloads requiring precise interactions.

Streaming:

- Data flows continuously in a unidirectional manner, moving from the slave to the master. This setup is ideal for situations where data is processed on-the-fly. As the name suggests, it is exceptionally well-suited for throughput-oriented processing where maximizing bandwidth is crucial. This type of interface is particularly suitable for various forms of stream processing, and it excels in scenarios where "bump-in-the-wire" processing is required, a common use case for modern FPGAs.
- It typically involves reduced overhead, particularly in terms of resources, as fewer wires translate to decreased resource usage and less congestion. Additionally, it retains optional signals (such as TDEST, TUSER or TID) that can be effectively used for any necessary

identification or routing needs. Furthermore, the streamlined initiation setup can often result in lower latency in many situations, making it an excellent choice, especially for bulk processing tasks.

- The streams have to be initiated somewhere and this is typically handled by DMA engines running in the background. While the overhead of DMA initiation is minimal during large-scale transfers, for fine-grained random access patterns, the overhead can become quite substantial.

So which of these two interfaces better fits Coyote?

Coyote primarily concentrates on a "bump-in-the-wire" style of processing. This choice is primarily driven by the proven effectiveness of FPGAs in these specific use cases. Moreover, the underlying infrastructure that Coyote primarily uses, such as high-performance network stacks for smart NIC operations, also operate in a stream-based manner [184, 187], and it complements DMA-style operations quite well. This DMA-style approach to computation provides the benefit of programming simplicity, as application developers can readily interact with the handshaking logic for both initiating transfers via descriptors and managing data.

When all of these factors are considered together, streams emerge as a distinct preference for the data movement within Coyote. They present a notably simpler interface in contrast to full memory-mapped buses, which in this context do not have many advantages. Additionally, they effectively reduce the number of signals that traverse the system and, in the end, are exposed to the user kernels. Simultaneously, the evident advantage of memory-mapped interfaces for configuration-related tasks within the system (often managed by the host CPU) means that we utilize memory-mapped buses for comprehensive system management.

3.1.2 The DMA Wrapper

With the decision to construct the data movement interface around streams for both DMA-initiating descriptors and actual data, coupled with the choice to orient the complete management layer around memory-mapped buses, the next step is to identify a suitable core that can deliver this functionality with reasonable resource overhead while ensuring high performance. The previously mentioned PCIe integrated block encompasses low-level interfaces, and it requires a DMA wrapper to facilitate higher-level (AXI level) interaction with it. Instead of building it from grounds up, we choose to employ AMD's provided XDMA core, which perfectly aligns with the functionalities needed for envisioned Coyote's interfaces.

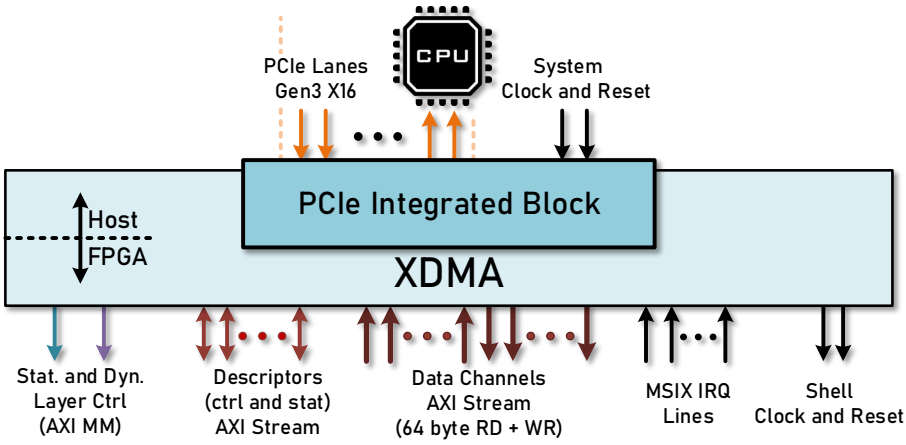

Figure 3.4: Coyote configuration of the XDMA core

The AMD XDMA core [34] is primarily made to facilitate efficient data transfer between the host CPU memory and the FPGA fabric. Essentially, it functions as a DMA wrapper layered atop a PCIe integrated block, which is hardened within the FPGA. The core's standout feature is its provision of a DMA interface to the FPGA fabric, along with a range of user-configurable options that can be tailored to suit the specific requirements of the target system. The configured XDMA core and its interfaces, used in Coyote, is shown in Figure 3.4.

Operating at a default frequency of 250 MHz, the core features a 64-byte data interface, yielding a theoretical maximum bandwidth of approximately 16 GB/s.

Additionally, this core supports up to four concurrent stream or memory-mapped channels, with the caveat that only one of the types can be selected at a time — no combination of both is possible. Data movement within these channels is controllable both from the host CPU and from the FPGA, through a dedicated bypass interface, to which descriptors of the transfers can be passed (through a stream interface) directly from the FPGA fabric.

The bypass interface holds significant importance in the context of Coyote, as it allows the implementation of a shared-memory model. It enables FPGA applications to initiate data transfers without any reliance on the CPU or the host OS kernel for intervention. It's worth noting that these descriptors operate using physical addresses and typically contain information about the address itself, in addition to the associated length of the DMA transfer.

The core also includes robust support for interrupt management, implementing the MSI-X (Message Signaled Interrupts eXtended) technology. MSI-X enables devices to directly send interrupt messages to the CPU through the PCIe, reducing overhead and enhancing system perfor-

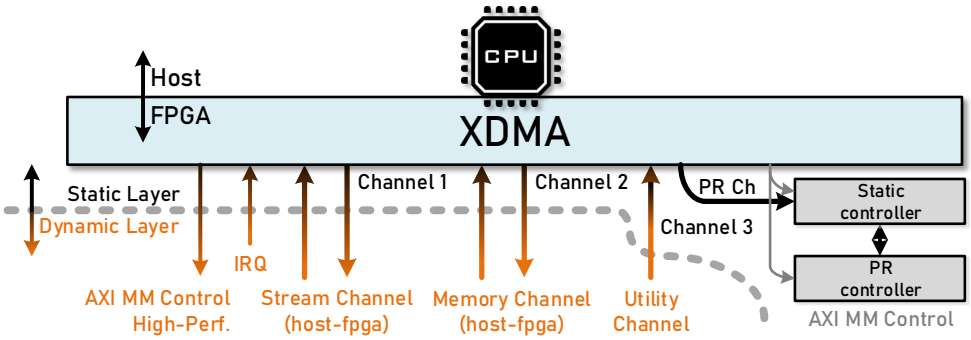

Figure 3.5: Coyote shell interfaces between static and dynamic layers

mance. This feature is especially beneficial for high-performance devices with numerous interrupt sources, such as FPGAs. Given the multitude of interrupt sources in Coyote, including user notifications, page faults, migration completions, reconfiguration, etc. , the MSI-X's capability to efficiently manage a large number of interrupt sources becomes absolutely crucial. Furthermore, MSI-X is well-equipped with operating system support to efficiently handle these interrupts within device drivers.

3.1.3 The Shell Interface

Coyote exposes various interfaces to facilitate communication and data transfer between different components in a system. These interfaces are closely tied to the configuration of the DMA (Direct Memory Access) wrapper, which is responsible for controlling and managing data movement. We will now explain these interfaces (shown in Figure 3.5) and their connection to the DMA wrapper's configuration.

3.1.3.1 Management of the Shell

Memory-mapped access is the preferred method for management operations because most management tasks involve writing to configuration registers or configuration memory. The DMA wrapper provides an AXI bypass memory-mapped bus within the FPGA fabric, which is accessible from the CPU via one of the exposed PCIe Base Address Registers (BARs). These BARs define memory-mapped regions and I/O ports for PCIe-connected devices, enabling communication with the CPU and system memory. They essentially tell devices where their memory is located.

A significant advantage of the XDMA core's AXI bypass bus is its generous 64-byte size. This is particularly beneficial because it can be accessed not only through standard 32-bit or 64-bit read and write operations but also via intrinsic AVX (Advanced Vector Extensions) instructions. AVX is a specialized set of instructions designed to enhance modern CPUs' processing capabilities. It extends the CPU's instruction set architecture to work with wider registers, often 256 bits wide, allowing simultaneous processing of multiple data elements. This parallelism is particularly beneficial for Coyote because it allows for faster data transfers to the FPGA fabric. With AVX instructions, the CPU can pass larger amounts of data with fewer read or write operations, resulting in improved efficiency and lower overall latency. This primary AXI bypass bus serves as the main CPU configuration channel. After leaving XDMA, the data path traverses the static layer and then advances to the dynamic layer, where it is employed for all shell configuration tasks. This interface is later multiplexed within the dynamic layer to accomodate a range of configuration spaces.

In addition to the primary AXI bypass bus, XDMA provides a separate AXI Lite bus accessed through a separate BAR. The AXI Lite protocol is a simplified version of AXI, designed for straightforward and lightweight communication primarily used for configuration purposes. In Coyote, we leverage this AXI Lite bus to manage the descriptors used to initiate dynamic reconfigurations. Since these operations involve significant overhead, the use of a high-performance bus for initialization is unnecessary, making the AXI Lite bus perfectly suitable for the task. Control of this interface is managed within the Coyote device driver, where this interface is memory mapped (`ioremap`) into the kernel's virtual address space.

3.1.3.2 Host to FPGA "Streaming" Channel

Host CPU-FPGA communication is crucial for swiftly exchanging data between the compute units in the FPGA and the host memory. These direct "streaming" interfaces eliminate the need for additional data copying to local FPGA memories. Many state-of-the-art shells, including ones deployed in the cloud [208, 30], often lack such interfaces, which necessitates the initial data copy to the FPGA memory. This copying incurs significant overheads, particularly when processing requires only one pass of the data. It complicates the pipelining of such processing and adds to the overhead.

For this reason, Coyote has a dedicated channel which directly moves the data between the user operators in vFPGAs and the host memory. This channel contains the high-performance data bus and the accompanying descriptor. The fields provided within each descriptor used for initiating

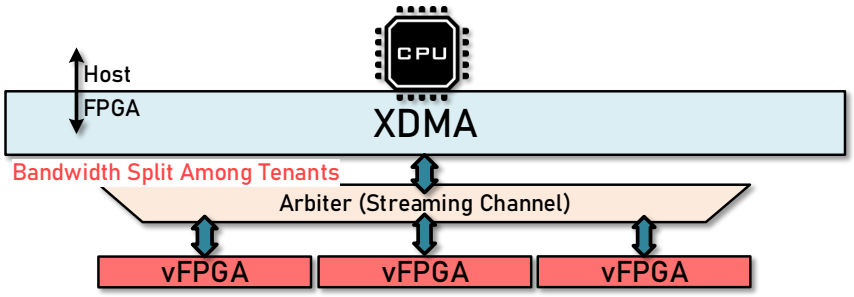

Figure 3.6: Streaming channel.

the transfers are shown in the Table 3.1. Along with these fields the descriptor contains the usual valid and ready handshake signals. The data channel is a conventional 64-byte AXI stream channel functioning at the native 250 MHz frequency of the DMA wrapper. Subsequently, this channel is multiplexed to all of the vFPGAs located within the dynamic layer of the shell.

All descriptors used with this channel at the end include physical addresses. These addresses are obtained by passing through the dedicated memory management unit in the dynamic layer and through its internal TLBs. As a result, all accesses made by user applications running in Coyote are done using virtual addresses, ensuring the system's protection is properly maintained. This is made possible by the virtualization layer and is a key reason why this channel is typically absent in state-of-the-art commercial shells (lacking virtualization) that are currently available.

The conventional use of this channel is for stream-oriented processing, where operations usually involve minimal passes over the data. It is extensively utilized for bump-in-the-wire processing and is essential for achieving high performance, especially in networking operations such as RDMA, where avoiding additional data copies is critical.

Table 3.1: Host DMA descriptor structure.

Signal name	Size [bits]	Description
`paddr`	40	Starting physical address within the host memory
`len`	28	Length of the transfer
`last`	1	Last flag, determines the last packet (assert tlast on the last beat)

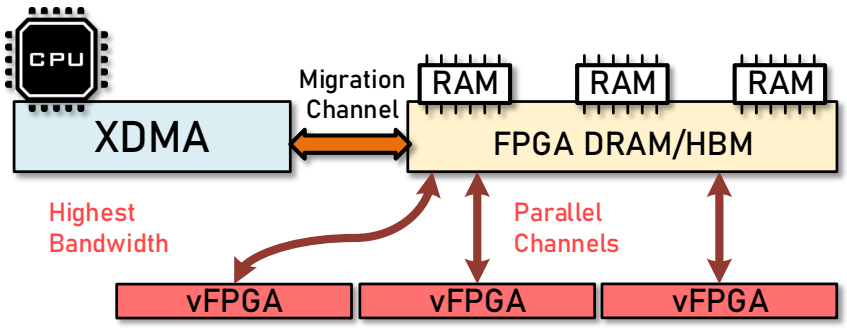

Figure 3.7: Migration channel.

3.1.3.3 Migration Channel

From an organizational perspective, this channel shares the same structure as the streaming channel, featuring identical descriptors and data interfaces. It is primarily employed when FPGA-side memory, whether it's DRAM or HBM, is in use. This usage scenario typically pertains to large-scale iterative processing models, as seen in applications like machine learning training. Its primary purpose is to facilitate high-speed data transfer between the host and FPGA memory.

Since migrations are closely tied to the memory model utilized within Coyote, this channel operates under the complete control of the Coyote kernel driver. While users can initiate transactions toward/from the FPGA-side memory, migrations are managed from the host. It is important to note that this channel is active only if the FPGA's memory stack is enabled; otherwise, it is tied-off within the dynamic layer. Additionally, the descriptors for this channel do not pass through the MMU layer as they are controlled from the host kernel which obtains the necessary translations.

Migrations are typically operations which are not on the critical path, whereas the streaming channel is (transfers data directly for application hardware). Ideally, the distribution of bandwidth between the streaming and migration channels should favor the streaming channel. However, XDMA lacks native support for managing bandwidth ratios between channels, and both channels are given the same priority. An alternative approach in Coyote is to use a single host channel multiplexed into these two channels. The custom multiplexer within Coyote prioritizes the streaming channel. While this option can optimize bandwidth allocation for the critical path, it does come at a slight overall bandwidth cost due to the reduction in available XDMA channels (it does offer some resource usage advantages). It's important to note that this is not the default

configuration in the supplied static layer of Coyote but rather an optional configuration.

3.1.3.4 Utility Channel

The third and final channel used in Coyote is the *utility channel*, exclusively reserved for system management operations. Unlike the other two channels, this one serves system-level functions, and while it is bidirectional, the two directions have distinct purposes.

Bitstream Loading

The data stream from the host to the card serves the purpose of loading bitstreams stored in the host DRAM. These bitstreams are subsequently transmitted to the reconfiguration controller.

Bitstream loading, much like migrations, is under the complete control of the host Coyote driver, eliminating the need for translation of virtual addresses (handled within the kernel module). These bitstreams are usually loaded in larger chunks (typically 2 MB in size), which means they don't require an excessive number of descriptors transmitted from the host. Therefore, they are loaded via the AXI Lite bus, which manages the configuration slave registers in the static layer.

It's important to note that the throughput of this controller is significantly lower (typically reaching up to around 1 GB/s), compared to the PCIe bandwidth. As a result, this channel has a limited impact on the overall distribution of PCIe bandwidth.

Writeback Functionality

The other direction in this utility channel serves the purpose of the *writeback* functionality within Coyote (illustrated in Figure 3.8). Writeback functionality enables the CPU to periodically check the host memory for completion status counters. These counters are updated whenever transfers are completed, signifying successful data delivery to/from the end users. This mechanism is crucial because it facilitates efficient polling. User applications are no longer required to use the PCIe to check for completion events, reducing the overall PCIe bandwidth wasted.

By default, the XDMA core supports the writeback feature and comes with internal registers for configuring read and write counters for each channel. These counters are linked to specific locations in the host memory. However, in Coyote, the situation is more complex. Not only do we have completions for the streaming channels, but also have dedicated channels between the vFPGAs and the FPGA memory, which are not exposed to host DMA and are unrelated to PCIe. Similar story applies to the network completion events. In Coyote, we need to keep the host counters updated for all these services, in addition to monitoring any migrations that may have taken place.

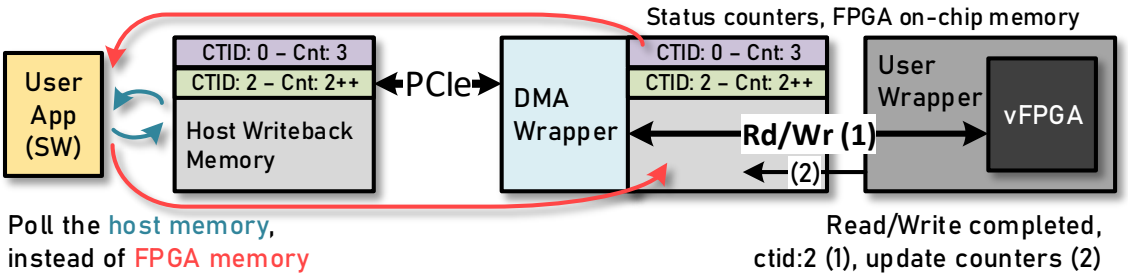

Figure 3.8: Writeback functionality

Furthermore, one of Coyote's notable features are *Coyote threads* running concurrently in each of the vFPGAs. Each thread of execution possesses a unique identifier and may belong to an entirely different user process. Consequently, we need to separate counters for each of these threads. To accommodate this added complexity, Coyote employs a separate channel (card to host) to maintain up-to-date completions for all transfers and all services, across all Coyote threads. These updates involve writing only single 4-byte values (counter values) back to the host, which utilizes only a small portion of the total PCIe bandwidth.

3.1.3.5 Interrupts

Interrupts are essential in computer systems because they enable real-time responsiveness. In the context of Coyote, interrupts are crucial as they serve as the primary mechanism for informing the host CPU about significant events happening within the FPGA. Coyote utilizes a variety of interrupts to cover critical system events, including the detection of page faults within the MMU, invalidation events and completion events, particularly for operations that involve longer processing times (TLB invalidations, migrations, dynamic reconfiguration, etc.). These interrupts ensure efficient handling of key events within the system.

The interrupts can originate from both the static layer and the dynamic layer. The primary purpose of the interrupt from the static layer is to inform the host CPU about the completion of dynamic reconfiguration. This process can be time-consuming, typically taking milliseconds, making polling highly inefficient and wasteful of CPU cycles. In the dynamic layer, interrupts from all the vFPGAs are consolidated. Parallel lines are utilized to allow efficient concurrent processing of interrupts for all vFPGAs in the host Coyote driver.

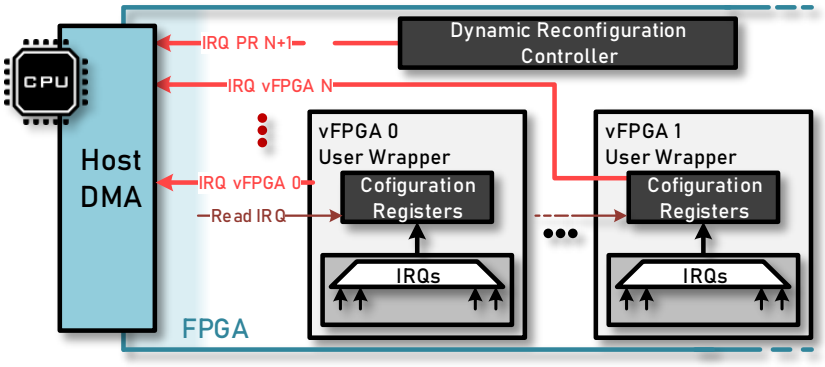

Figure 3.9: Interrupt lines within Coyote

The DMA wrapper core supports up to 32 interrupts, but only up to 16 of them are exposed to user logic in the FPGA. The remaining interrupts are reserved for internal wrapper functions like descriptor completions.

In the context of Coyote, where multiple parallel hardware processes (vFPGAs) are in use, and each vFPGA can generate various interrupts, careful management of these 16 available interrupts is essential. To avoid running out of interrupt lines, multiplexing is employed within each vFPGA. The interrupts in each vFPGA are multiplexed based on predefined priorities. This approach ensures that only one interrupt line is used for each vFPGA. To identify the type of interrupt and any additional information it carries, configuration registers present in each vFPGA are read from the CPU. This configuration allows for parallel processing of vFPGA interrupts within the kernel module, optimizing the management of interrupts at the OS level.

Upon receiving the interrupt, the driver initially reads all the information contained in the interrupt and promptly proceeds to handle it. Explanation of the interrupts within Coyote are given below and are ordered by descending priority:

1. **Migration completions** (offload and sync operations) - These interrupts are triggered upon the completion of a migration process when they wake up the waiting thread (Listing 3.1). Migrations involve transferring pages from the host to the FPGA-side memory (offload operation) or moving data in the opposite direction (sync operation). These interrupts are assigned the highest priority because they are directly related to data movement. While migrations are generally not on the critical path, they remain closely linked to application execution.

2. **Invalidation completions** - These interrupts inform the host Coyote driver that the TLBs have been updated, indicating the successful completion of the latest invalidation request from the host. While TLB invalidations may seem straightforward, they can have quite a few nuances and carefull handling is necessary. It's crucial to wait for any "in-flight" transfers related to the pages currently being invalidated to be completed before proceeding with invalidation in the host OS MMU. Hence, interrupts are crucial in this scenario since this operation can be time-consuming.

3. **Page faults** - Page faults are a fundamental element of Coyote's shared virtual memory model. Given the asynchronous nature of this operation, interrupts are crucial for proper management (Listing 3.2). Interrupts play the vital role of notifying the driver about page faults and provide essential information such as addresses, length, and identifiers for each page fault. Subsequently, based on this information, the driver can initiate various scenarios to resolve them, including potential migrations, TLB updates, or the insertion of new mappings.

4. **User notifications** - This interface enables user applications in hardware to inform software applications running in the user space about specific events occurring in the vFPGA. It serves as an arbitrary notification mechanism available to hardware applications, allowing them to send asynchronous messages to the user space. As this is only a notification mechanism, as such it has the lowest priority.

Listing 3.1: Waking up the thread that sleeps for completion.

```
1 // Completion of an offload operation (migration)
2 atomic_set(&d->wait_offload, FLAG_SET);
3 wake_up_interruptible(&d->waitqueue_offload);
4 return IRQ_HANDLED;
```

Listing 3.2: Schedule a work queue task.

```
1 // Page fault, deffered work
2 irq_pf->d = d; fpga_read_irq_pfault(d, &irq_pf);
3 INIT_WORK(&irq_pf->work_pfault, fpga_pfault_handler);
4 return IRQ_HANDLED;
```

The interrupt is managed through the Interrupt Service Routine (ISR). This ISR operates within the interrupt context, a dedicated state where the CPU suspends its current task to address the interrupt. It's critical for this context to be brief; the service routine should efficiently enter, examine the interrupt status, schedule the appropriate work queue, and exit promptly, all without

invoking any sleep functions. Subsequently, the hardware is promptly notified that the interrupt has been handled, and the worker thread executes the callback function within the device driver that resolves the interrupt at a deferred time. This approach ensures efficient interrupt handling with minimal disruption to ongoing processes.

3.1.4 Device Driver

The host device driver is a vital element in the Coyote system. It is a software component that operates within the Linux kernel, serving as an intermediary between the computer's operating system and the FPGA, including its full array of peripherals. Essentially, the driver acts as the interface that enables communication and interaction between user applications running in the user space on one side and the hardware applications in vFPGAs on the other side. During runtime, the driver manages the entire FPGA device, handling various tasks such as memory mappings, dynamic memory allocations, page faults, and partial reconfigurations. This comprehensive driver-side management ensures the seamless operation of the overall system.

After compiling the driver, the first step is to insert it into the host operating system as a loadable kernel module. This action must be performed by a privileged user. This insertion follows the loading of the bitstream image containing the static layer of the shell and a subsequent rescan of the interconnect. Once the interconnect is initialized inside the driver, it is able to be used to read the loaded configuration within the FPGA. Both the configurations of the static layer and the currently loaded dynamic layer are read. It's worth noting that the dynamic layer can be swapped during runtime, and these changes are initiated through a system call. The driver handles all the intricate aspects of this process, making it possible to reload the shell without the need to re-insert the driver or take the system offline.

Crucially, the driver sets up all the required device structures to enable and support the operation of all virtual FPGAs. Additionally, for each vFPGA, the driver creates a separate character device file. These device files serve as a communication bridge between the user space and the driver. This communication is enforced through traditional system calls like: *open*, *close*, *mmap* and *ioctl*.

The base device structure in the driver is the structure built around the interconnect. In a PCIe attached device this structure will be initialized by the *probe* function that identifies and initializes the hardware device when it is detected in the host CPU root node. It plays a crucial role in the device discovery process and allows the kernel to match the FPGA with the Coyote driver.

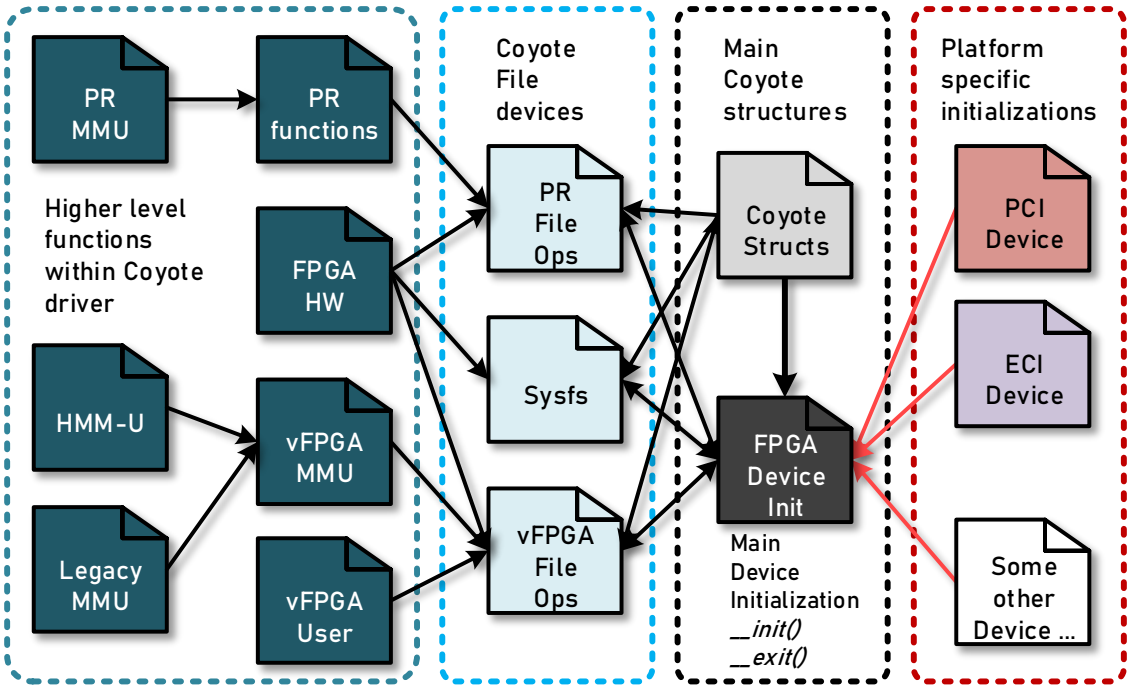

Figure 3.10: Device driver architecture and platform modularity.

An important consideration during the design of the Coyote driver was to create it in a way that minimizes its dependency on the specific underlying platform and interconnect being used. This design approach was crucial to ensure the portability of the overall system. The driver's architecture is illustrated in Figure 3.10. It can be seen that the main driver data structure encapsulates the underlying platform, while the interface to the rest of the driver code remains independent of these specifics, thereby guaranteeing a high degree of portability.

3.1.5 Cross-Platform Compatibility

The platform interface presented in Coyote, as discussed in previous sections, is intentionally designed to be platform-agnostic. Although we have made certain protocol choices, such as adopting the AXI protocol, this protocol is widely recognized in the hardware industry and inherently adaptable to various platforms. This adaptability allows us to seamlessly integrate these interfaces into different hardware environments as required.

While our current solution involves using the XDMA wrapper core from AMD on top of PCIe as it satisfies all feature requirements, it's essential to emphasize that we are not bound by it.

The XDMA core does come with a range of additional features, such as scatter/gather descriptor chaining, external control through AXI slave interfaces from the FPGA fabric, internal writeback functionality, memory-mapped data movement, etc. However, most of these features heavily rely on the XDMA driver provided by AMD. In Coyote, we do not use this driver; instead, we rely on our own driver that implements a Coyote-specific set of features. From the core's feature set, we rely on the descriptor bypass for transfer initiation, the memory-mapped AXI control for management purposes, and the MSI-X interrupt support.

This means that the XDMA core can easily be replaced by a custom DMA wrapper tailored for Coyote's functionality. This wrapper could then be integrated on top of the platform-specific interconnect, offering flexibility and adaptability to different hardware platforms.

3.2 Enzian

To demonstrate Coyote's versatility on various platforms, we successfully ported it to our in-house developed Enzian system. Enzian [70, 101], shown in Figre 3.11, is a purpose-built computer designed for systems software research. It combines a robust server-class CPU with a sizable FPGA in an asymmetric NUMA (cache-coherent Non-Uniform Memory Access) configuration.

The key feature that sets Enzian apart is its high-performance, low-latency ECI (Enzian Coherency Protocol) interconnect. In contrast to traditional PCIe-based shells that prioritize throughput but introduce latency due to PCIe overheads, the ECI protocol is a native inter-socket cache coherence protocol designed to seamlessly integrate with the host ARM CPU. This unique design ensures that the FPGA is closely and efficiently connected to the CPU, going beyond the role of a standard PCIe-attached device. Instead, the FPGA operates as an additional NUMA node, elevating it to the status of a first-class citizen in this heterogeneous system.

3.2.1 A Research Computer

Key emphasis during the Enzian design process was a no-compromise approach to features. Enzian, architecture depicted in Figure 3.12, is a two-socket asymmetric system. One node features a 48-core Marvell Cavium ThunderX-1 ARMv8 CPU operating at 2.0 GHz. This chip additionally includes various on-die accelerators, such as cryptography and networking components with internal switching capabilities.

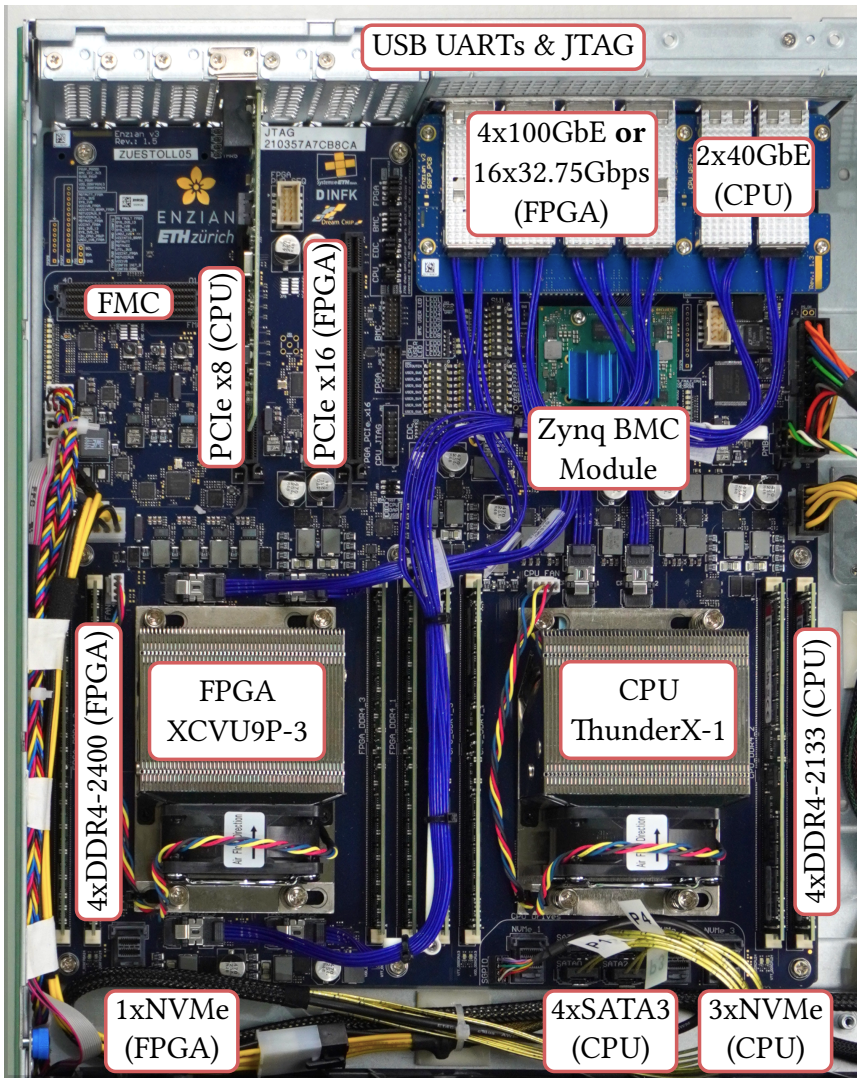

Figure 3.11: Enzian board.

The CPU is equipped with 128 GiB of DRAM distributed across four DDR4 channels. On the reconfigurable side, the second node is a Xilinx XCVU9P Ultrascale+ FPGA. This FPGA boasts four DIMM slots with collective support for up to 1 TiB of DDR4 DRAM. It's connected to the CPU via the CPU's native coherent interconnect, using 24 10 Gb/s lanes with a total theoretical bandwidth of 30 GiB/s in each direction.

Enzian's abundant networking capabilities make it particularly suitable for networking tasks. The CPU node provides two 40 Gb/s Ethernet interfaces, while the FPGA node offers four 100 Gb/s Ethernet links. This extensive bandwidth on the FPGA side opens the door to intriguing research possibilities, going beyond the traditional usage of the FPGA as a smart NIC. Further-

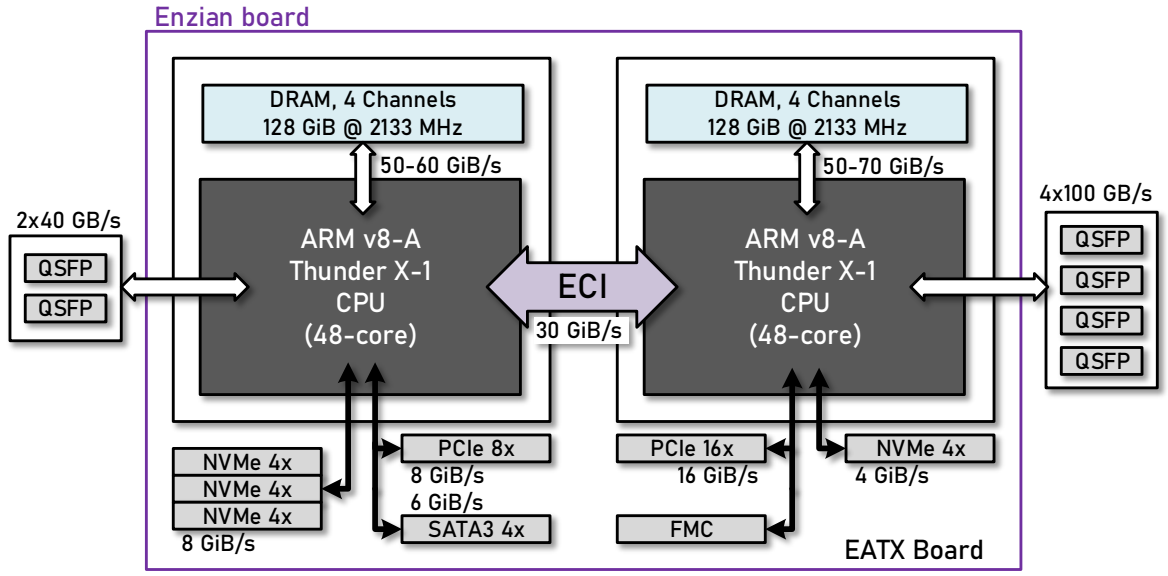

Figure 3.12: Architecture of the Enzian computer.

more, the FPGA node includes a PCIe x16 slot and a single NVMe connector, allowing the addition of extra devices, such as external accelerators or storage devices.

Enzian's research-oriented design offers an additional advantage in the form of comprehensive built-in monitoring system support. Enzian incorporates a dedicated measurement circuit capable of monitoring and controlling voltage regulators across all power rails with remarkable precision. This capability holds significant value for FPGA research, particularly in addressing one of the frequent assertions within the FPGA community—namely, the potential for lower power consumption and increased efficiency when compared to other accelerators [173, 149]. Enzian's extensive monitoring support serves as a valuable tool for assessing and substantiating these claims.

Moreover, it's worth noting that Enzian is open-source, which means it offers a rare and expansive platform within the research community that can be thoroughly explored without constraints or cost.

3.2.2 Cache Coherent Interconnect

A key feature of Enzian in comparison with commercial hybrid CPU/FPGA server platforms is that CPU and FPGA are connected via the CPU's native inter-socket cache coherence proto-

col, rather than PCIe. In compairson to PCIe based systems, this closer coupling between the CPU and the FPGA within Enzian provides a much lower latency which benefits finer grained workloads to a much higher extent.

The Enzian Coherence Interface (ECI) is structured around the MOESI protocol, featuring 128-byte cache lines. This protocol enables cache lines to be cached on either the home node or the requesting node. Furthermore, it provides support for non-cached small I/O reads and writes, along with inter-processor interrupts. These features are of great importance in the context of Coyote as they are utilized to establish the aforementioned interfaces Coyote shell requires. It's important to note that the system's physical address space is statically allocated between the CPU and FPGA. This allocation allows us to structure the FPGA address space in a manner that aligns with Coyote's internal organization.

Within the Enzian project, additional work was done to enable FPGA applications to directly interact with a cache coherence protocol in a more flexible manner than simple coherence assumptions. This work [175] implemented directory controllers, which abstract much of the protocol complexity, offering a portable interface for application logic to access memory coherently alongside the CPU and, importantly, interact with the CPU's Last-Level Cache (LLC).

This solution consists of hardware components that can be instantiated on an FPGA to provide a simplified interface for user logic, coupled with an OS kernel module that allows convenient access from CPU software. The key concept was to modularize and simplify the complexity of the coherence protocol into scalable and reusable hardware units.

The primary goal of these directory controllers is to facilitate cache-coherent FPGA and CPU-side memories. The fundamental concept here is to enable additional processing to be pipelined on top of the coherent FPGA-side memory, where various acceleration circuits could be explored. However, this approach is designed for scenarios where the CPU primarily manages transactions on a cache-line basis.

This model, however, differs from the one employed in Coyote, which aims to provide mastering capabilities on both the CPU and FPGA sides. This FPGA-side mastering capability is a significant advantage in Coyote, as it (along with the overarching shared memory model) enables the integration of complex stacks requiring FPGA side mastering capabilities, such as the RDMA network stack. For this reason, Coyote does not use directory controllers on Enzian, as their general memory model does not align with Coyote's.

Instead, Coyote's interfaces are integrated on top of our custom DMA wrapper, which is built on the low-level Enzian ECI link. This bidirectional link utilizes 32 lanes, twice the number of

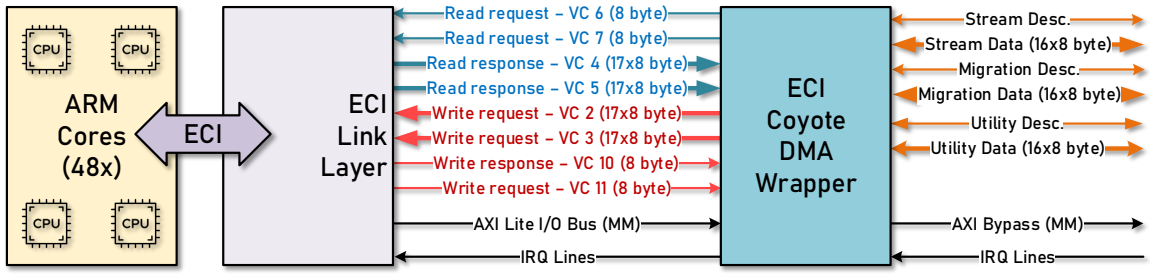

Figure 3.13: Enzian-Coyote interfaces.

transceivers required for traditional PCIe integrated block on AMD cards. An essential feature is that all accesses made from Coyote are fully coherent with the host CPU caches. This, along with lower latency and higher throughput of the link provides Coyote with a powerful advantage in comparison to traditional PCIe-based systems when running on Enzian.

Ultimately, the development of the DMA wrapper, the integration of Coyote's AXI interfaces, and the adaptation of Coyote's device driver served as a valuable test to assess the portability of Coyote to other platforms.

3.2.3 Coyote on Enzian

We won't delve into the inner working of the ECI link architecture. For Coyote, what matters most is the interface this link provides. This interface is depicted in Figure 3.13 and is structured around the concept of virtual channels. Coyote utilizes eight of these virtual channels, with four serving as request channels and the other four as response channels. The ECI consists of two parallel links, and to maximize bandwidth, both links need to be utilized in conjunction.

The read or write command transmitted through the read request or write request channel is a single 8-byte word. In the read request channel, only this word is present, while in the write request channel, the word is accompanied by the cache line that is being written with the request. This command is issued through a standard AXI stream interface and, therefore, is accompanied by the combination of valid and ready handshake signals. The command's structure is illustrated

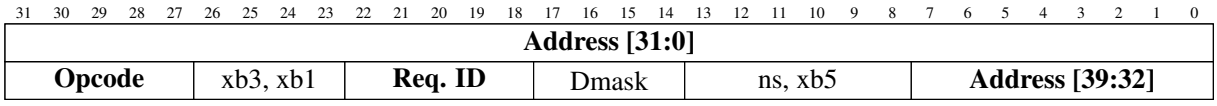

Figure 3.14: Enzian request command bit layout (64-bit)

in Figure 3.14. The important fields, for Coyote's operation, within the request command are the *opcode*, the *address* and the *id* fields.

The *opcode* field is utilized to instruct the ECI with a specific operation. The proprietary coherency protocol contains a vast number of different operations with different caching behaviors that can be exploited. In Coyote, we rely on the two operations shown in the Table 3.2.

Table 3.2: ECI read and write operation codes

Opcode	Operation	Description
`MREQ_RSTT`	Write	Remote store immediate, don't cache anywhere
`MREQ_RLDT`	Read	Remote immediate

These operations facilitate data movement in both directions and are sufficient for all host data operations in Coyote. Both of these operations respect the current caching state in the CPU cache hierarchy. This means that if writes are in process, data within them will not be cached if it's not already in the cache. However, if the data is already in the cache, it will be updated, thus preserving the cache coherency on the host CPU side. The same applies to reads; the read data will be the most up-to-date, including the data residing within the CPU caches. Therefore, no flushing or other explicit updates to the CPU caches are needed.

The *address* field in the ECI protocol contains the physical address of the data in the host memory. This field is 42 bits wide, with the most significant two bits indicating the target node. The ECI protocol has the flexibility to connect up to 4 NUMA nodes together, but since Enzian is a two-node machine, these two bits can be disregarded within Coyote. The remaining 40 bits represent the actual DRAM offset within the host node.

This address needs to adhere to a specific aliasing pattern, which is employed to enhance the cache locality of data in the CPU caches. The aliasing process takes place within the L2 cache, so all accesses originating from the FPGA node must have their addresses aliased accordingly. This modification should be applied to all the descriptor addresses provided by Coyote. The operation involves only simple XOR operations and can be easily done combinatorially. The code representing the address field conversion is shown in Listing 3.3.

The *id* field in the ECI protocol is a 5-bit field used to assign a specific identifier to each cache-line transfer sent to the ECI. This field is important because there are no guarantees regarding ordering within the ECI. As a result, the responses to subsequent transfers can be reordered freely. This out-of-order operation is a necessity within the ECI to achieve any meaningful per-

formance. The ECI allows the users to issue multiple in-flight read or write requests, rather than just rely on slow sequential transactions. As indicated by the 5-bit field, up to 32 reads and 32 writes can be issued independently, with each request tied to a particular ID. The corresponding response will have the same ID for identification purposes.

Listing 3.3: The aliasing of ECI physical addresses within Coyote.

```
// Address aliasing
always_comb begin
aliased        = 0;
aliased[32:13] = non_aliased[39:20];
aliased[12:8]  = non_aliased[19:15] ^ non_aliased[24:20]; // XOR operations
aliased[7:5]   = non_aliased[14:12] ^ non_aliased[27:25];
aliased[4:3]   = non_aliased[11:10] ^ non_aliased[24:23] ^ non_aliased[13:12];
aliased[2:0]   = non_aliased[9:7]   ^ non_aliased[22:20] ^ non_aliased[14:12];
end
```

However, the out-of-order operation provided by the ECI in most cases is not that useful for end users, particularly in scenarios where applications rely on strict ordering guarantees (which is coincidentaly the case for most user applications running in FPGAs). While Coyote offers the option for heavy interleaving of transfers, it ensures that transfers within a single descriptor are always maintained in order. Therefore, a reordering stage, tracking all in-flight transfers, is needed. This *id* field is used to implement this operation in Coyote.

Each individual in-flight transfer within the ECI has a fixed size, equal to a single cache line, which is 128 bytes. This cache line size differs from the traditional 64-byte cache lines commonly used. Coyote was designed to accommodate these 64-byte cache lines, and all the buses are appropriately sized. Moreover, the 64-byte data buses align well with the limits of the current FPGA fabric potential, such as the transceiver interfaces (e.g., PCIe, QSFP) that are typically designed to handle data rates of 100 Gbps. System clocks in modern FPGAs typically range from 250 to 350 MHz, and these buses are sized to efficiently utilize the available bandwidth for these frequencies. To optimize the performance of these 100 Gbps interfaces, the appropriate bus size can be roughly estimated as follows:

$$bus\ size\ in\ bits \approx \frac{100\ [Gb/s]}{300\ [MHz]} = 100\ [Gb/s] \cdot 3.33\ [ns] = 333\ [b]$$

This equation considers the maximum link utilization, and achievable real-world throughput is typically lower [95]. The rough estimate from this equation is 333 bits, thus considering the difference between practical and theretical bandwidths, it's safe to assume the rounded size of 512 bits, making it an appropriate and commonly used bus width for modern FPGA designs.

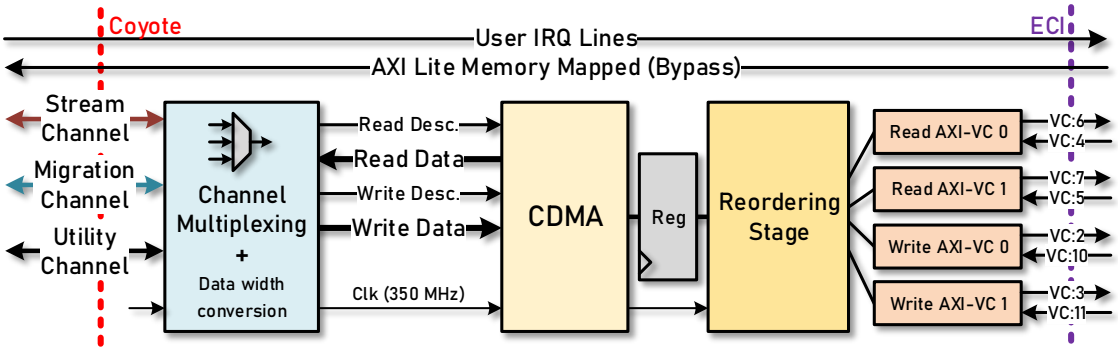

Figure 3.15: Hih level overview of the Enzian DMA wrapper within Coyote.

This is also the exact size of the bus found in PCIe hardened block for AMD FPGAs, aligning with practical design considerations.

On Enzian, the 128-byte bus width follows the cache line size of the ECI. It thus maximizes performance from the two 64-byte links. The ECI specification provides a maximum theoretical bandwidth of $\approx$ 23.8 GB/s, although in practice, the achievable bandwidth is slightly lower. The 128-byte bus is indeed overprovisioned, as theoretically, this bus operating at 300 MHz is capable of achieving nearly 38 GB/s.

To optimize routing resources and reduce congestion in Coyote, the decision was made to use a traditional 64-byte bus and perform the necessary data width conversion. To approach the optimal bandwidth of the link, Coyote has to be used with a slightly higher system clock. Operating at 350 MHz allows Coyote to come close to saturating the ECI link, as will be demonstrated in the benchmarks (Section 3.2.5).

3.2.4 Enzian DMA Wrapper

The Enzian DMA wrapper is designed to bridge the gap between the out-of-order virtual channels in the Enzian ECI and the descriptor-oriented AXI interfaces used in Coyote. The architecture of this DMA wrapper is illustrated in Figure 3.15. The DMA wrapper is fully pipelined, ensuring that it doesn't introduce any bandwidth related bottlenecks. While it does add some latency, Coyote primarily targets throughput-oriented workloads, so the slight additional latency is not a significant concern. The DMA wrapper is the sole hardware module within Coyote that exhibits platform-specific characteristics.

Arbitration Stage

The first stage within the DMA wrapper, which interfaces with standard Coyote shell, is an arbiter stage where the three existing channels in Coyote are multiplexed into a single ECI wide channel. Arbitration is handled using a priority scheme where the highest priority is given to the streaming channel, followed by the migration and utility channels. Since this stage is before the reordering stage, the arbiter can assume that, at this point, all transfers will be executed in-order. This, along with the fixed arbitration scheme, simplifies the arbiter design, which is based on queuing up the channel requests for all of the in-flight transfers and then relying on the information within these queues to process the responses. After the arbitration is performed, the channel is passed into a dedicated data width converter, which converts the channel size from 64-byte to 128-byte, as used by the ECI link.

The DMA Engine

The next stage within the wrapper is the actual DMA stage where the conversion between the AXI stream protocol and full AXI memory-mapped interface is performed. As explained earlier, Coyote is based on the notion of a streaming interface, while the ECI link operates with cache-line granular accesses that are memory-mapped. The conversion between the stream channels and their descriptors to the memory-mapped AXI bus is thus needed at this stage. For this purpose, we implement a DMA engine that effectively handles the protocol conversion between these interfaces. The core is highly parametrizable and provides high throughput transfers with a parameterized number of in-flight requests, as well as support for cache line unaligned transfers.

At this stage, once the requests leave the DMA engine, they are traditional AXI memory mapped-requests. Each request can be a burst request, which means that multiple incremental cache lines might be requested at once. This burst capability is used to enhance data transfer efficiency.

Register Slicing

It's worth noting that additional register slices are a crucial part of the overall pipeline structure. This practice is not unique to this context but is a common feature in Coyote and FPGA designs in general. The purpose of these register slices is to minimize inter-module wiring lengths, making it easier to meet stringent timing requirements.

The strategic placement of these slices is of paramount importance, particularly in multi-SLR (Super Logic Region) FPGAs like the ones Coyote runs on. In these multi-SLR FPGAs, individual FPGA dies, or SLRs, are interconnected via specialized, resource-limited pathways. As a result, optimizing register placement near these interconnections is essential to minimize delays and resource usage. Although these slices add some latency (typically 1 - 3 clock cycles per

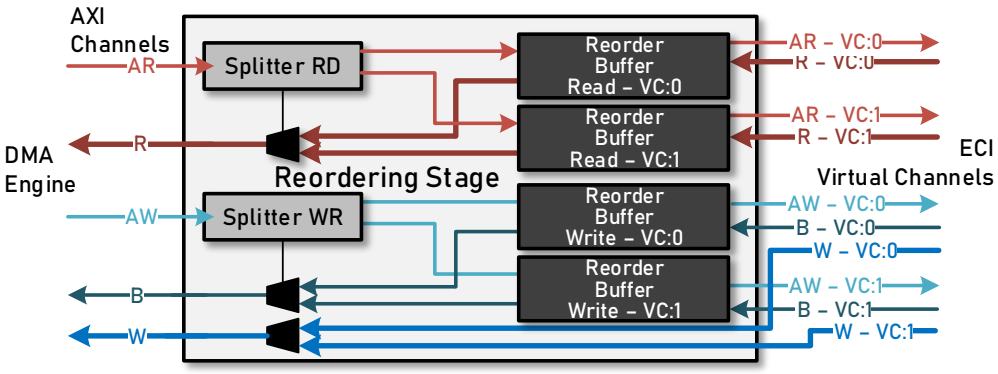

Figure 3.16: Reordering stage

slice, depending on the configuration), they don't impact the bandwidth and do not introduce any bubble cycles.

Reordering Stage

The subsequent reordering stage is the central processing stage responsible for adapting the protocol between Coyote and ECI, shown in the Figure 3.16. This stage's function is to take incoming AXI requests, which may contain multiple bursts, and split them into individual memory-mapped requests, which are then directed to separate request channels. It's important to note that both read and write operations have two parallel virtual channel interfaces that can be utilized concurrently thus typically two individual requests for each of these will be generated at a time.

The primary purpose of the reordering stage is to keep track of all in-flight requests and allocate a unique identifier to each of them. This supervision of requests and their corresponding identifiers is managed within a circular reorder buffer, as depicted in Figure 3.17. The role of this buffer is to supervise the out-of-order execution of read or write requests, ensuring that their sequencing is preserved. This guarantees that responses are sent back to the shell in the correct order, even if the requests were executed out of order to enhance performance.

The circular reorder buffer comprises 32 entries for each virtual channel, allowing for a total of 64 operations to be outstanding for both reads and writes, given that there are two parallel virtual channels for each. Additionally, this buffer serves the crucial function of providing backpressure to upper layers when necessary.

The memory-mapped AXI bus entering the reordering stage includes extra signals not present in the lower ECI stack layers. For instance, after each read operation, the last transfer must have

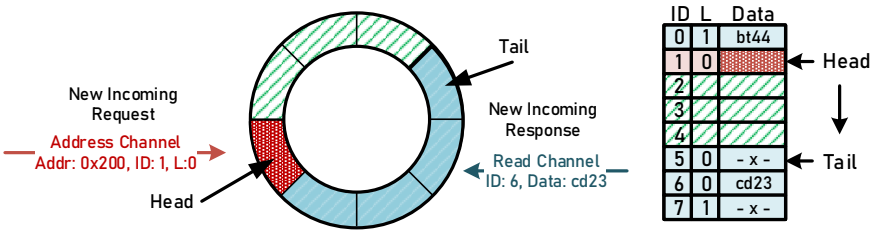

Figure 3.17: Reordering buffer

its last flag asserted. However, the ECI does not provide this specific information. Therefore, this supplementary information (along with any other) is stored within the circular buffer and is attached to every response. The operation of the reorder buffer is shown in the Figure 3.17.

Virtual Channels

In the final stage, conversion to virtual channel interface takes place. Although the official AXI bus protocol is still used, it no longer contains advanced signals such as *burst*, *cache*, *rlast*, etc. This is because all transactions essentially become single load/store instructions at this point.

The only remaining tasks are to package the address information, the identifier information into the virtual channel headers, and provide data if the operation is a write. All other functions, such as managing the virtual channels, arbitration, and coordination with additional ECI messages, are handled by the lower layers of the ECI stack.

Other Interfaces

Regarding other interfaces used by Coyote, such as interrupts and the management memory-mapped (bypass) bus, these are (to some extent) provided by the ECI stack.

For example, interrupts are supplied in a raw physical format rather than a message-signaled format. While message-signaled interrupts similar to traditional MSI-X do exist within the ARM cores on Enzian, they have not been extensively explored at the time of writing this thesis. Consequently, Coyote can currently only rely on the physical interrupt lines exposed by the ECI to trigger the interrupt controllers on ARM cores.

As for the memory-mapped management (bypass) bus, the current version of Enzian offers only an 8-byte wide AXI Lite bus. This, along with the large address space exposed in the FPGA, is more than sufficient to handle all configuration requirements within Coyote. However, it's important to note that this bus has lower throughput compared to the full-fledged AXI bus supplied by the XDMA core on AMD platforms. The reason for not implementing a full AXI bus within

the ECI core on Enzian is that AVX instructions needed to enhance the bus throughput are not supported on Thunder-X ARM cores in the same manner and through the same libraries as on x86 architectures. Similarly to interrupts, SIMD instructions are available in these ARM cores, but come in their own flavour and integration of these libraries into Coyote was not yet explored, leaving this as potential future work for Enzian's Coyote port.

Driver Adaptation

As discussed in the device driver section (Section 3.1.4), minimal changes are required in the overall driver architecture to ensure compatibility with the ECI on the Enzian platform. This adaptability is primarily due to the driver architecture's modular design, which separates critical driver functions from platform-specific elements, offering a platform-agnostic interface for higher-level functions.

There are, however, a few notable distinctions on the Enzian platform which require examination. First, Enzian lacks support for message signaled interrupts, which are extensively utilized in the base Coyote driver for interrupt registration and detection. On Enzian, only physical interrupt lines are available (for now), and their limited number places constraints on the driver. The optimal approach to handling interrupts on the Enzian platform is an area that might receive (hopefully) more attention in the future.

Second, there is an absence of the conventional AVX high-performance bus on the Enzian platform. This, however, does not impact the design of the driver extension, as this functionality is exclusively used from user space, where dedicated memory regions are memory-mapped for direct access. Therefore, no driver-level adaptation is necessary.

It's important to note that the Linux kernel's support for PCIe offers a comprehensive array of well-established functions and structures for PCIe-attached devices, encompassing probing, device detection, registration, memory mapping, memory allocation, and more. However, when working with the ECI link, which lacks the extensive support of PCIe (and, in fact, lacks broader system software support in general), developing a platform-specific device driver extension becomes a more intricate task. Some functionalities present in PCIe, such as automatic hardware detection, dynamic driver loading/unloading, driver binding, power management, and error handling, are not directly available. As a result, the driver extension is implemented at a more rudimentary level and relies on a simpler interface to interact with the device, mainly through global structures via an I/O-mapped memory region.

A similar situation could potentially arise with many new interconnect technologies that are emerging, such as CXL, CCIX, QPI, UPI, and others. While these interconnects offer hardware

advantages and improved performance, their complexity and the absence of robust software support might pose significant challenges when implementing higher-level systems and applications on top of them. It's thus essential to think about the gap between cutting-edge hardware and well-established system software to maximize the potential of these interconnects.

3.2.5 Link Evaluation

In this section, we will conduct a performance comparison between Coyote on the Enzian platform and a conventional PCIe-based Alveo data-center card [15]. The objective is to demonstrate the advantages of the ECI link, which offers low latency and high bandwidth, and to validate Coyote's adaptability to different platforms. This evaluation also aims to establish Enzian as a viable research platform for heterogeneous systems. Through this evaluation, we will also reaffirm how the standardized interfaces in both hardware and software play a crucial role in ensuring the portability of the Coyote system.

This test aims to compare the performance of the ECI link to the traditional PCIe link while using the same Coyote shell configuration on both links.

It's important to note again that ECI utilizes two parallel links, each having a theoretical bandwidth of around 12 GB/s. They contain 24 lanes in total. In contrast, PCIe operates with only one link (16 lanes), which is not bifurcated. Initially, it might seem that this configuration provides an unfair advantage to Enzian, but it's important to acknowledge that this distinction is inherent to the design of these interconnects.

In this benchmark, we will assess both read and write operations, obtaining both the latency and throughput measurements across a range of transfer sizes. Specifically, we will benchmark the FPGA's read and write operations using uncached, coherent, cacheline-sized transactions, all occurring over the ECI link to the host CPU memory. Notably, all transfers are initiated from the host CPU, thus the latency measurements include communication overheads. For this reason the latency graphs do not show the minimal latencies of the links.

We will conduct a comparative analysis between the Enzian platform and an AMD Alveo u55c data center card [15], using identical benchmarks for both platforms. The Alveo card is integrated into an Intel Xeon server, utilizing a 16-lane PCIe Gen3 connection that provides a maximum theoretical bandwidth of 16 GB/s per direction. Our measurements will involve eval-

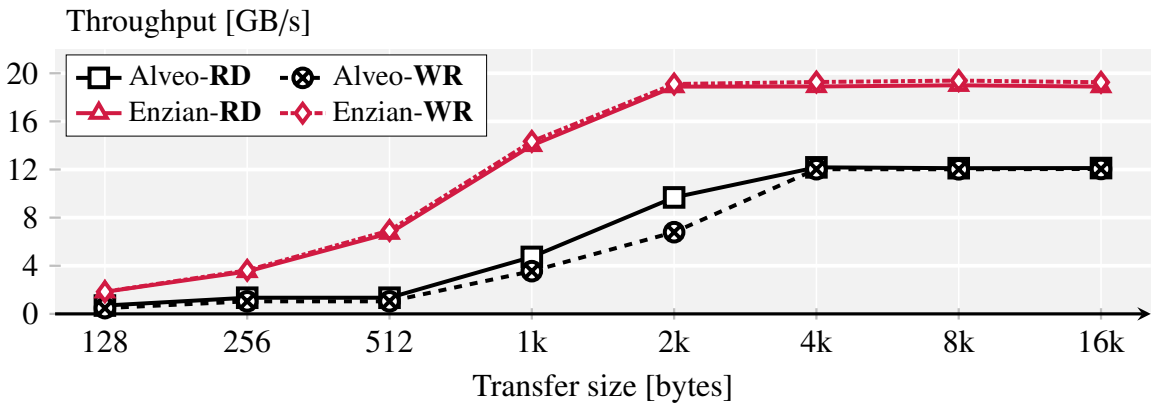

Figure 3.18: Throughput comparison between ECI and PCIe links in Coyote.

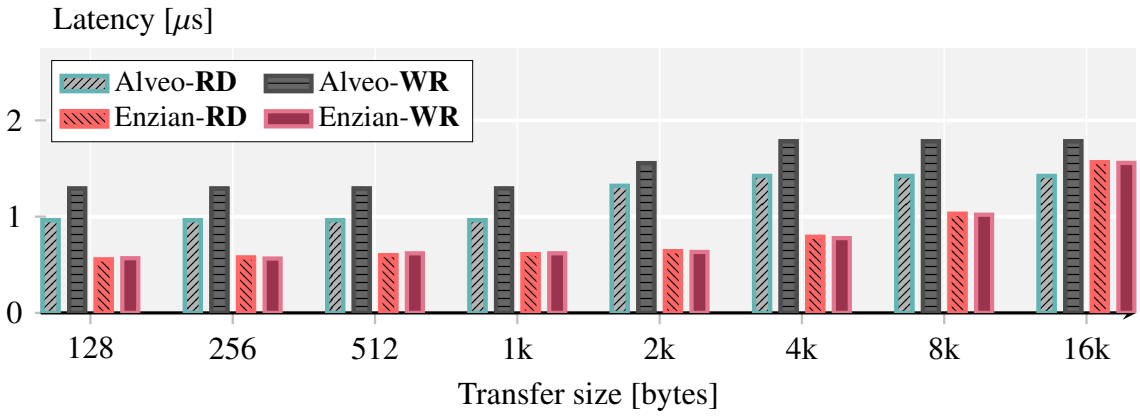

Figure 3.19: Latency comparison between ECI and PCIe links in Coyote.

uating the real data throughput and latency for different transfer sizes, with each measurement being based on an average of 10,000 runs.

Figure 3.18 illustrates the comparison of read and write throughputs. Enzian consistently surpasses the PCIe-based baseline platform in both read and write operations. The pronounced advantage of the higher bandwidth link is apparent and offers significant benefits for workloads requiring direct access to host memory. Furthermore, the increased bandwidth enables the support of a greater number of vFPGAs and user applications, with each application enjoying a larger share of the total bandwidth. It's essential to highlight that for smaller transfer sizes, Enzian proves to be a more efficient platform as well, even in batched workloads, primarily due to its lower latency.

Additionally, the overall latency (as shown in Figure 3.19) for ECI, is approximately half that of PCIe. Notably, large ECI transfers consist of a sequence of essentially independent low-latency

cache line transactions. In contrast, PCIe is designed for throughput, involving an upfront cost to set up large transfers. ECI's superior performance in cache-line sized transfers is particularly valuable for applications that do not involve extensive bulk data transfers, such as fine-grained graph processing algorithms.

It's also worth examining the differences in resource consumption between the Enzian platform and traditional PCIe devices. In this part of the experiment, we compare Enzian, which employs the xcvu-9P FPGA chip, to Coyote running on an AMD vcu118 development board [20]. This board was chosen because it features the same FPGA chip as Enzian (same resource distribution, but with a different speed grade), providing the most accurate basis for comparison.

Table 3.3 presents the FPGA resource consumption of the Enzian DMA wrapper combined with the ECI core, as well as the XDMA wrapper alongside the PCIe integrated block. It's important to note that the PCIe integrated block is a hardened core, meaning it doesn't utilize any of the FPGA fabric. In contrast, the ECI lacks such hardening and relies entirely on the FPGA fabric.

The fact that the ECI link utilizes significantly fewer resources (almost half of the CLBs) while delivering a performance advantage is surprising. This can likely be attributed to the additional features of XDMA, most of which are not actively utilized in Coyote. This serves as further validation that exploring a custom DMA wrapper for Coyote on Alveo platforms should be considered.

The overall results highlight some of the benefits that can be obtained by adopting a new design perspective. By considering the FPGA as being on par with the CPU, Enzian can attain distinct performance trade-offs that can greatly advantage numerous application classes.

Significantly, these experiments demonstrate that Coyote is portable and can operate on a variety of platforms with highly distinct interconnects that lack many common characteristics.

Table 3.3: DMA wrappers - resource usage comparison.

Hardware Platform	CLBs	LUTs	Registers	Block RAM
Enzian DMA wrapper	7322 (4.95 %)	46 742 (3.95 %)	30 241 (1.28 %)	71.5 (3.31 %)
vcu118 XDMA wrapper	14 106 (9.5 %)	69 403 (5.87 %)	64 773 (2.74 %)	96 (4.44 %)

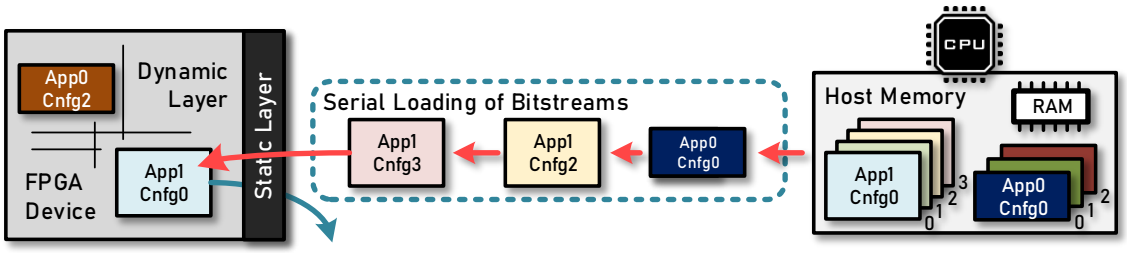

Figure 3.20: Partial reconfiguration of user applications.

3.3 Hierarchical Dynamic Reconfiguration

Now, let's return to the architecture of the static layer and explore the second major module within it, the *dynamic partial reconfiguration*. This feature enables the temporal sharing of the FPGA, and presents one of the most substantial advantages these devices possess, adding large level of flexibility (especially when compared to traditional ASICs).

Partial reconfiguration allows specific regions of the FPGA to be modified or reconfigured while other parts of the device continue to operate without disruption. This is a departure from full reconfiguration, where the entire FPGA is reprogrammed. In the case of full reconfiguration, the entire system must be taken offline as critical infrastructure within the FPGA is reinitialized. This incurs significant overhead and may necessitate additional actions on the host system, sometimes even requiring cold reboots. Clearly, this is an impractical scenario in cluster environments within data centers.

In the context of Coyote, partial reconfiguration plays a crucial role. It is used for any user-initiated loading of the shells and is also employed for dynamically loading applications independently. This loading of applications is shown in Figure 3.20. Full reconfiguration is reserved for loading the initial static layer. In a cluster environment, this full reconfiguration step is performed only once (excluding maintenance periods), and the Coyote static layer remains active throughout the cluster's usage. As a result, there is no downtime during the exchange of shells (dynamic layer) or applications (application layer). These other layers can be swapped independently at any point in time.

The size of the dynamic layer's bitstream is usually close to the size of the entire FPGA, excluding the static layer, which constitutes a relatively small portion of the device. Application bitstreams, on the other hand, vary in size based on the specific applications and user floor-

planning. Typically, application bitstreams are considerably smaller and can be loaded more swiftly.

The entire sequence of steps involved in loading bitstreams in Coyote includes:

- Bitstream generation: All bitstreams are generated using Coyote's build system (described in Section 3.3.5). There are three distinct types of bitstreams that can be generated:
    1. Full configuration bitstreams, which include the static layer, are exclusively generated by Coyote maintainers and are typically released in specific versions. Nevertheless, since Coyote is open-source, users have the flexibility to make modifications to these bitstreams if required. Separate versioning of these bitstreams allows for the loading of additional hierarchical layers, regardless of the origin of the static layer bitstream.
    2. Dynamic layer (user shell) bitstreams contain the dynamic layer and initial user applications. These bitstreams constitute the initial hierarchical level and are therefore generated once for each version of the user shell. This bitstream remains consistent for all the various applications that can run within this shell configuration.
    3. Application layer bitstreams (vFPGAs), represent arbitrary user applications. Multiple configurations of these bitstreams can coexist for each vFPGA, all within the same version of the overarching shell.
- Loading the bitstreams involves distinct procedures for the different types of bitstreams:
    1. The initial static layer bitstream is loaded by a privileged user during system initialization, typically using the JTAG (Joint Test Action Group) port. This operation cannot be performed through partial reconfiguration and necessitates taking the system offline.
    2. Dynamic layer bitstreams are usually loaded by a privileged user through a dedicated Coyote shell loader program, which initiates the required system calls to the Coyote driver.
    3. Application layer bitstreams are loaded by Coyote scheduler operating in the user space. Requests to load these bitstreams can be supplied to Coyote service daemon operating on the host (typically through a Unix domain socket). These requests can originate from non-privileged users. Important point to note is that the necessary bitstreams must be preloaded in the Coyote service's library, which contains validated bitstreams. The validation process is left to the discretion of Coyote users.

- The driver initiates the process of transferring the bitstream to the FPGA configuration. It does so by pinning the pages where the bitstream is stored and commencing the transfer of these to the FPGA through the CPU-FPGA interconnect.
- The bitstream loading process begins. It's important to note that at this stage, the dynamic region undergoing reconfiguration is undefined. This means that any logic it drives is also undefined. Consequently, measures are put in place to protect the rest of the system, specifically the higher hierarchical layers, from any unintended signals originating from this undefined region. To achieve this, a comprehensive layer of decoupling is applied, effectively isolating all propagation of signals from the dynamic region.
- Once the reconfiguration process is finalized, the new bitstream becomes active in the designated region of the FPGA. The FPGA smoothly transitions to utilizing this new configuration. It's worth noting that this bitstream is not considered active until the entire loading process is completed. Until loading is fully finalized, it essentially functions as a shadow configuration. If corrupted bitstream is loaded or the loading is not completed, this temporary configuration is discarded. Upon switching from the old to the new configuration, the old bitstream is no longer utilized.

Before delving into the hierarchical nature of Coyote's reconfiguration, let's first take a look at the architecture of the reconfiguration controller that orchestrates it.

3.3.1 Reconfiguration Controller

The reconfiguration of the FPGA is handled by a centralized hardware primitive called ICAP (Internal Configuration Access Port) [13]. It provides the interface that allows the dynamic reconfiguration of portions of the FPGAs, while the rest of the device continues to operate.

ICAP establishes a connection with the FPGA's configuration memory, which houses bitstreams defining the configuration of various FPGA logic elements, including CLBs, registers, BRAM, URAM, DSP blocks, etc. Within ICAP, internal data pathways facilitate the targeted transmission of bitstream data to specific regions of the FPGA fabric, ensuring dedicated routing to its intended destinations (typically SRAM configuration cells).

Controlled by a set of registers and dedicated control logic, ICAP supervises the entire configuration process. These registers take charge of managing the configuration sequence, defining write addresses for the data, and orchestrating the orderly execution of configuration operations.

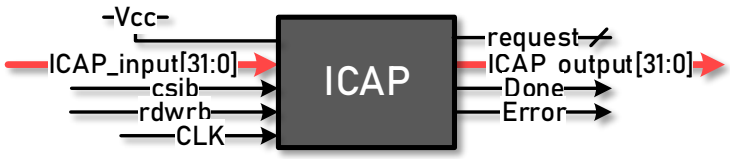

Figure 3.21: ICAP controller interfaces

Additionally, ICAP offers an external interface that follows standard protocols like JTAG, enabling external devices or tools to engage in and regulate the reconfiguration procedure. While it's commonly used in tandem with AMD's configuration tools and libraries for streamlined reconfiguration, ICAP also serves as a versatile interface, accommodating integration with various user defined controllers and tools as needed.

The Figure 3.21 illustrates the ICAP interface, which boasts a variable data width that can range up to 32 bits. This interface encompasses both input and output data ports, serving as entry and exit points for loading and reading bitstreams, respectively. When loading bitstreams, they are processed in 4-byte segments, requiring proper ordering of bytes within these data words. It's crucial to note that ICAP employs the most significant bit ordering, a vital detail to bear in mind since default produced bitstreams are not typically generated in this order.

Furthermore, traditional bitstreams with a `.bit` extension come with headers that should not be fed into ICAP. Consequently, a modification of the produced bitstreams is essential, either through hardware or software, where specific instructions can be used during compilation to ensure the correct format for ICAP.

Notably, ICAP lacks any backpressuring mechanism, placing the full responsibility for managing data flow on the external controller. This is achieved through the `csib` input port, which can be driven low to indicate that the data on the input bus is valid at the current clock cycle. The additional `rdwrb` port signals whether a read or write operation should be executed. The ICAP also signals the completion, along with any errors if present, through dedicated `Done` and `Error` output ports, respectively.

While ICAP's read operation enables users to access the current bitstream, offering the potential to capture the FPGA's current state, it's important to recognize that bitstreams primarily consist of a raw sequence of bytes, and the information represented by these bytes is often proprietary. Consequently, these read operations may not provide substantial benefits at this stage, and fully capturing the FPGA's state remains a challenging venture. This challenge is further compounded

when dealing with additional complexity, as seen in cases like Coyote, where the required state may be distributed throughout the FPGA (network stacks, memory stacks, etc.).

One crucial aspect of partial reconfiguration is its speed, which is of paramount importance in systems like Coyote. In such systems, reconfiguration speed is a critical factor, given that this process occurs sequentially. In a multi-tenant environment especially, a slow reconfiguration speed can significantly and adversely affect the overall system performance.

FPGAs typically offer various methods for loading bitstreams, but many of these methods have inherent limitations on the achievable speeds. For instance, the default approach available to users within the Vivado toolchain is to utilize an AXI HWICAP core [31] for bitstream management. While this core allows for FPGA reprogramming with bitstreams loaded through a traditional AXI Lite configuration and can be controlled by user logic within the FPGA, it falls short of harnessing the FPGA's maximum reconfiguration speed potential. This limitation is primarily due to the slow bus and the sequential nature of the process, where bitstream words are written with a notably low throughput.

Additionally, the MCAP (Media Configuration Access Port) interface [3], is designed for PCIe-connected data center cards, similar to those utilized in Coyote. MCAP loads bitstreams via PCIe and, like the previously mentioned methods, relies on single data word writes, leading to similar throughput limitations. Another option, worth noting is PCAP (again with similar limitations), which is specifically tailored for embedded on-die CPUs within System-on-Chip (SoC) architectures.

Partial bitstreams can also be loaded through JTAG, but this method is typically reserved for full configurations and debugging processes. Moreover, the speed of this port is also relatively slow.

Thus, a better controller, able to fully saturate the ICAP reconfiguration interface, is necessary. The maximum frequency of the Ultrascale+ devices Coyote runs on (including Enzian which also contains an Ultrascale+ FPGA) is 200 MHz. Thus the maximum achievable throughput of the reconfiguration is $\approx$ 800 MB/s.

To achieve these speeds, it's crucial to integrate a dedicated DMA engine designed solely for this purpose. A similar concept has been explored in the context of System-on-Chip embedded architectures [124]. However, in our specific case, we're not depending on a combination of on-chip lightweight ARM cores (or softcore CPUs) and on-chip memory. Instead, we're leveraging a high-performance host interconnect and the host memory.

in Coyote, the bitstreams are thus loaded through the PCIe (or ECI). These 800 MB/s constitute only the small part of bandwidth (in ECI this is less than 5%) and thus do not have a tremendous

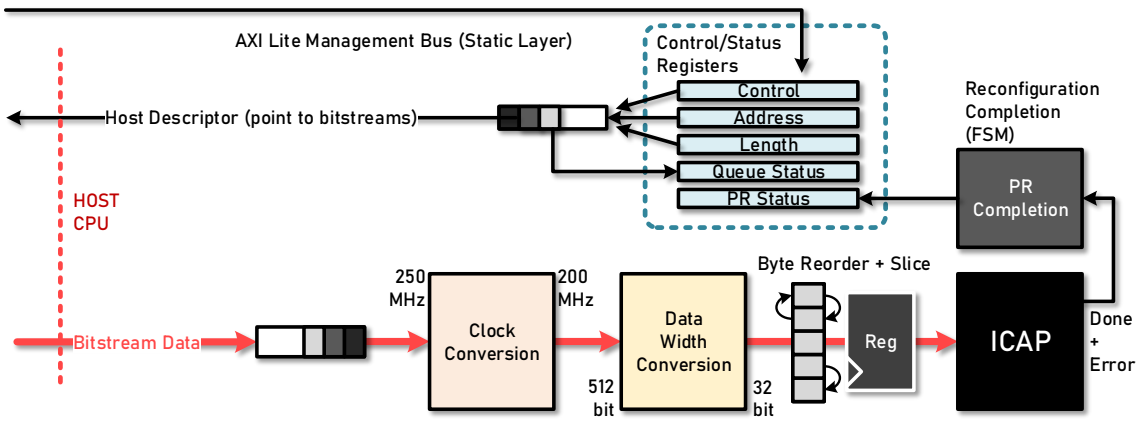

Figure 3.22: Coyote reconfiguration controller.

impact on the overall CPU-FPGA link performance. The bitstreams are fetched from the host memory. They enter the controller within the utility channel data stream and are queued up upon arrival.

Bitstream transfers are initiated from the driver, and the driver reconfiguration process is triggered from the user space. These system calls are accessed through a dedicated reconfiguration character device, created by the driver. All reconfiguration operations are handled via *ioctl* and/or *mmap* system calls. Interaction with the reconfiguration device file can only be done by privileged users.

As previously mentioned, there are two distinct bitstream loading events: one for the overall dynamic layer, which involves changing the entire user shell, and another for the application layer. Consequently, there are two separate system calls to handle these events. The primary distinction between these calls lies in the additional work required by the driver when reloading the dynamic layer, which involves reinitializing all internal dynamic data structures. Otherwise, the reconfiguration process within these system calls is very similar, with minor variations.

Typical sequence of steps begins with driver-level locking and decoupling. The decoupling process is selective, focusing only on the interfaces being reconfigured. Subsequently, the reconfiguration process is initiated by providing the virtual address and length of the pages containing the bitstreams. The driver uses the `get_user_pages` function to acquire the necessary bitstream pages from the user space. These pages are then pinned in preparation for the transfer to the FPGA. At this stage, the function enters a sleep state, facilitated by the use of mutexes, and awaits an interrupt event signaling the completion of the reconfiguration.

Listing 3.4: Invocation of the reconfiguration function in the `ioctl()` system call.

```
// Driver level lock and decoupling
mutex_lock(&d->rcnfg_lock);
pd->fpga_stat_cnfg->pr_dcpl_app_set = (1 << (uint32_t)vfpga_id);

// Reconfiguration
ret_val = reconfigure_start(d, vaddr, size, ctid);
// Completion
wait_event_interruptible(d->waitqueue_rcnfg,
    atomic_read(&d->wait_rcnfg) == FLAG_SET);
atomic_set(&d->wait_rcnfg, FLAG_CLR);

// Couple and unlock
pd->fpga_stat_cnfg->pr_dcpl_app_clr = (1 << (uint32_t)vfpga_id);
mutex_unlock(&d->rcnfg_lock);
```

Once the interrupt event occurs, the reverse process takes place. The pseudo code outlining these necessary steps, including locking, coupling, and other relevant operations, can be found in Listing 3.4.

In hardware, after exiting the input queues in the reconfiguration controller, bitstream data beats are initially routed to the clock conversion circuitry. This circuitry adapts the clock from the host frequency to the highest supported frequency of the ICAP. While it might be tempting to overclock the ICAP and use the interconnect clock beyond the vendor's specifications, we decided not to explore this route due to the costly nature of these chips.

The clock for the data streams is adjusted to 200 MHz, which is the maximum clock rate supported by all Coyote-compatible devices. The next step involves data width conversion, where the 64-byte stream is downsized to a 4-byte stream. It's here that the first signs of backpressure may manifest. Before entering the ICAP, the only other necessary operations involve hardware byte reordering and register slicing to trim the data paths. The ICAP then processes this data at its own pace, as long as the data is readily available. The comparison of the throughput of our custom controller with other available reconfiguration options is presented in Table 3.4.

Table 3.4: Reconfiguration throughput comparison

Reconfiguration Type	Max throughput [MB/s]	Interface
`AXI Hardware ICAP`	19	AXI Lite
`PCAP`	128	AXI
`MCAP`	145	AXI
`Coyote ICAP`	800	AXI Stream

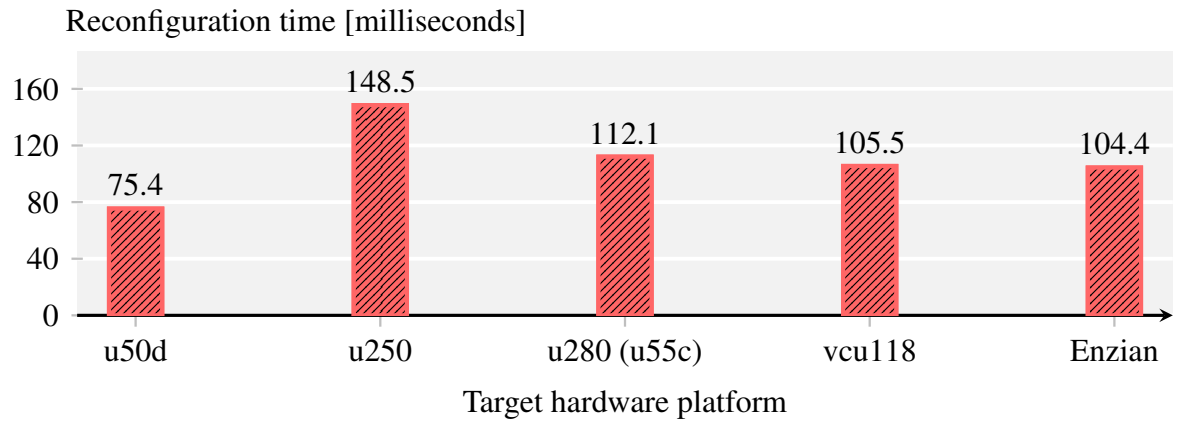

Figure 3.23: Dynamic layer (user shell) reconfiguration time on different hardware platforms.

3.3.2 Reconfiguration Performance

Now that we know the achievable performance, the question is how does this transfer to the real world applications and how much time do we actually need to reconigure these chips, both for the whole dynamic layer and for the applications within it.

The reconfiguration time is directly proportional to the size of the area being reconfigured. Since the static layer constitutes a relatively small portion of the overall shell (up to 10%) and remains fixed, the time required to reconfigure the dynamic layer remains nearly constant and is similar to the time needed to reconfigure the entire chip. However, this time also depends on the size of the target device. For example, certain chips, such as the u250 with 4 SLRs, require slightly more time than other chips (most other chips have 3 SLRs and a comparable amount of resources). Differences in reconfiguration times of dynamic layers (user shells) are shown in Figure 3.23, with smaller cards like the u50d exhibiting lower dynamic layer reconfiguration overheads. The reconfiguration times were averaged over 100 different runs. Certain cards, such as the u280 and u55c, share the identical chip layout. Consequently, their Coyote shells are also identical. Therefore, we conducted the benchmarks for only one of these chips (the one mentioned outside of the parentheses).

The conclusion from this test is that the swap time of the dynamic layer is in the order of 100 milliseconds, aligning with the achieved hardware throughput of 800 MB/s. The slight overhead, in comparison to the theoretical maximum, can be attributed to several factors, including the time needed for activating the new configuration, handling completion notifications through interrupts, and the additional kernel overhead.

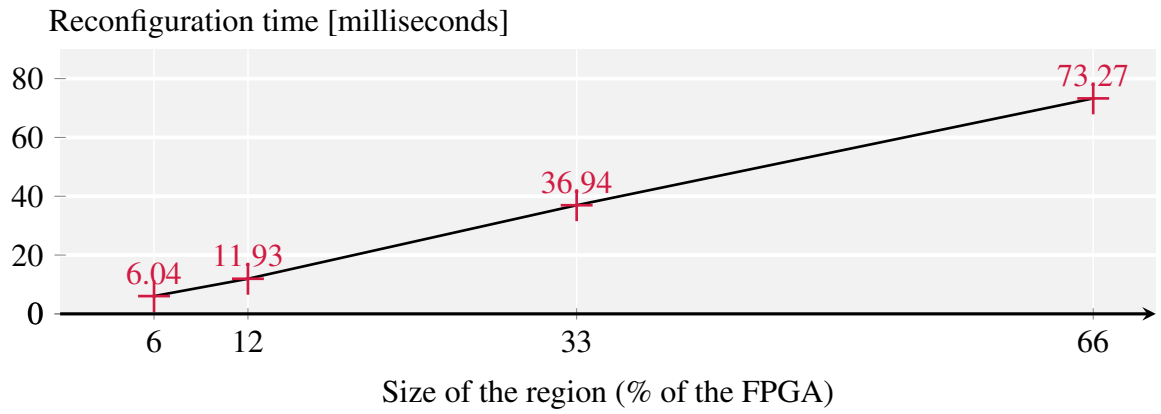

Figure 3.24: Reconfiguration time for vFPGAs depending on the portion of the chip assigned to them.

At the cost of this reconfiguration time, end users gain the capability to substitute the entire configuration of the shell and interchange complex services. For example, they could replace a TCP/IP stack with the RDMA stack to achieve higher performance data movement.

How does this reconfiguration time relate to applications? The key difference here is that, unlike the complete dynamic layer, these applications are arbitrarily sized and rely entirely on the users who manage their floorplanning. Consequently, the time it takes to reconfigure each of these applications is highly dependent on the specific floorplanning that has been carried out.

To understand the relationship between the size of these applications and the reconfiguration overhead, we conducted tests with different-sized virtual FPGAs (vFPGAs) within Coyote. For this test, as depicted in Figure 3.24, we utilized a *vu9p* chip found in the vcu118 development board and Enzian. We conducted floorplanning for the vFPGAs to align the pblocks (areas which are assigned to reconfigurable instances) with the percentage of the overall FPGA fabric. It's important to clarify that this does not imply that vFPGAs fully utilize the logic resources available within their designated pblocks. Rather, it's the floorplans that approximately match 6%, 12%, 33%, and 66% of the overall chip layout for these vFPGAs, respectively.

The results demonstrate that reconfiguration time scales proportionally with the reconfigured region's size, as anticipated. This underscores the significance of careful floorplanning. Grouping applications of similar sizes and starting with the most complex configuration is advisable. Careful consideration of routing resources is also needed. However, it's essential to recognize that application floorplanning wasn't extensively addressed in this thesis and poses a distinct challenge. Future research may explore whether automatic floorplanning can streamline this process and address existing issues.

Coyote's reconfiguration controller ensures swift and efficient reconfiguration. With a significant portion of bitstream handling managed in software (part in the kernel space and part in the user space), it provides a high-level bitstream management abstraction. This abstraction allows us to efficiently execute vital tasks such as arbitration, security, locking, decoupling, and file operations, all essential for the efficient and secure handling of bitstreams.

3.3.3 Scheduling

Partial reconfiguration on FPGAs remains a relatively slow process. It still takes on the order of milliseconds, which incurs a substantial penalty in comparison to conventional OS context switching costs [134]. Even with efforts to maximize the throughput of the reconfiguration controller (as done in Coyote), reducing this time to microseconds is not feasible due to hardware limitations, and large size of the bitstreams (up to 100 MB). Consequently, along with the inability to handle a proper preemption, treating partial reconfiguration as a simple context-switching mechanism is not feasible, and expectations must be adjusted accordingly.

In Coyote, scheduling is managed by a runtime manager in the user space, allowing it to make full use of high-level libraries to create advanced data structures, such as priority queues, which play a vital role in scheduling tasks. These tasks are submitted by users and contain information about a specific hardware operator. The hardware operator points to one of the available accelerators, which is stored in the accelerator library as a validated raw bitstream.

The scheduler's function is to initiate a reconfiguration request via a dedicated system call, and upon completion of the task, it notifies the user of the results. This process enables efficient task management and hardware acceleration within the Coyote framework.

The first optional scheduling stage in the system is the load balancer. When multiple vFPGAs are available, the load balancer aims to distribute requests among these vFPGAs. It accomplishes this by assessing the usage of current priority queues which are queuing up requests. Each vFPGA is equipped with one of these queues designed for the submission of tasks specific to that vFPGA. Importantly, this load balancing stage does not involve system calls and can be customized as needed to align with specific deployment requirements.

The priority queues reorder submitted requests based on two key parameters: 1) the priority of the request and 2) the operator ID. This modified priority-based queue scheme reorders requests in a way that places ones with the same operator ID and the same priority level adjacent to each

other. This rearrangement of requests facilitates grouping those that target the same hardware operator. As a result, there's no need for reconfiguring vFPGAs between these requests, assuming that the state within these hardware operators can be cleared. The aim of this strategy is to minimize the overall number of partial reconfigurations.

Upon retrieving a task from the queue, the initial step is to reconfigure the vFPGA. This is achieved through a system call, where virtual address and the size of the raw bitstream are passed to the driver. The driver subsequently acquires all the necessary locks to ensure that no other reconfiguration processes are underway and manages the reconfiguration operation. Once the reconfiguration is successfully completed, an event notification is sent back to the scheduler.

Subsequently, the user-submitted task can execute. During this execution, an additional user-space lock is held by the executing task. This lock is implemented as a named mutex in the shared host memory and serves to prevent other users from accessing the same vFPGA. Once the task execution is finished, the scheduler can proceed to the next task in the queue, maintaining controlled task execution.

We exemplify this scheduling concept through a straightforward example where we queue up tasks from four distinct applications (four different hardware operators). These applications are engaged in simple compute operations that essentially involve a loopback of data (from host to FPGA and back to host). Consequently, all four applications are fully pipelined, and all tasks are sharing the same priority. At the same time, we configure Coyote with three separate vFPGAs.

We dispatch tasks in three ways: 1) the "no-scheduling" approach is round-robin in both queues and vFPGAs. This causes a reconfiguration for each job and is specifically designed to be the worst-case scenario, 2) "random", as the name implies picks the next task from a random queue each time, and so has a 1-in-3 chance of needing a reconfiguration, and 3) "schedule" uses Coyote's heuristic of grouping tasks which share the same hardware operators.

Figure 3.25 shows total turnaround time for 10, 50, and 100 sequential tasks submitted to each of the four applications. In this case, each of these tasks is small in size and consists of a 4 kB data transfers. This scenario tries to mimick a typical multitasking scenario on a computer where multiple short-running burstable tasks are being context switched consistenly. Unsurprisingly, the figure clearly demonstrates that executing such tasks on FPGAs introduces considerable overhead, primarily attributed to partial reconfigurations. This is evident as, even with our basic scheduling technique, we can significantly reduce the overall execution times.

This implies that short-running tasks are not currently suitable for scheduling in modern FPGAs as they would be in a traditional operating system. However, this does not imply that scheduling

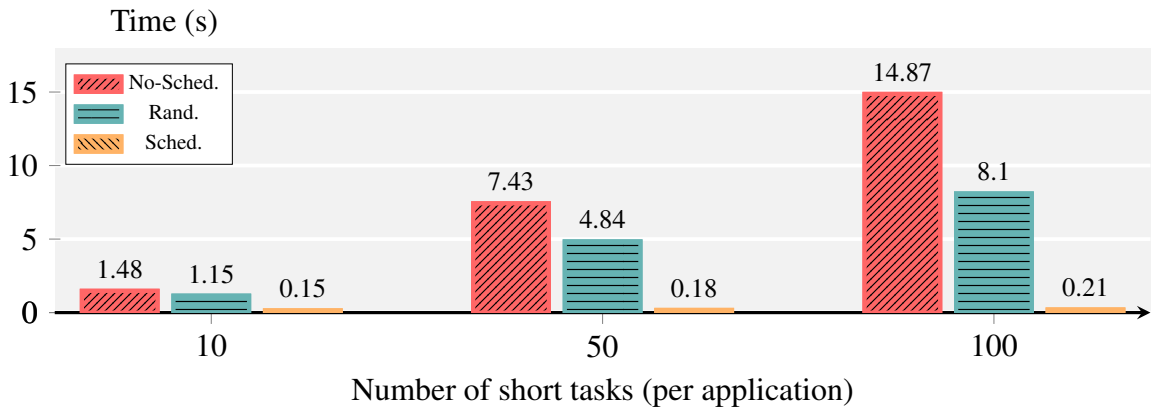

Figure 3.25: Scheduling performance for a) short running tasks and b) long running tasks.

is without merit. For considerably longer tasks, such as extensive video processing, stream processing, or large dataset inference, the combination of scheduling and partial reconfiguration can be highly advantageous, as the reconfiguration overhead in such scenarios becomes negligible.

Evidently, there is ample room for much more sophisticated task scheduling, beyond the scope of the research we've conducted. Crucially, Coyote, as a platform, provides the opportunity to investigate and advance scheduling techniques on a broader scale.

3.3.4 Nested Reconfiguration

The ability to implement hierarchical levels of reconfiguration is a fundamental feature that grants Coyote the adaptability needed for large, modern clusters that run diverse workloads in today's cloud-dominated landscape. This capability allows us to establish multiple levels of privileges within a single heterogeneous device.

For example, consider a cloud scenario where various accelerators can be dynamically loaded on demand. In this context, the cloud provider, within Coyote, can determine which type of shell configuration (dynamic layer) to load. These configurations might include different flavors of networking services or varying amounts of memory, and they can all be exchanged on-the-fly. Concurrently, a separate layer (application layer) remains dedicated to loading end-user applications. This functionality can even be exposed to the end-users. Furthermore, the cloud provider can decide how many user applications can run concurrently, which can be managed through various scheduling algorithms. This is just one example of a real-world deployment that can benefit from the hierarchical architecture of the Coyote shell and the flexibility it provides.

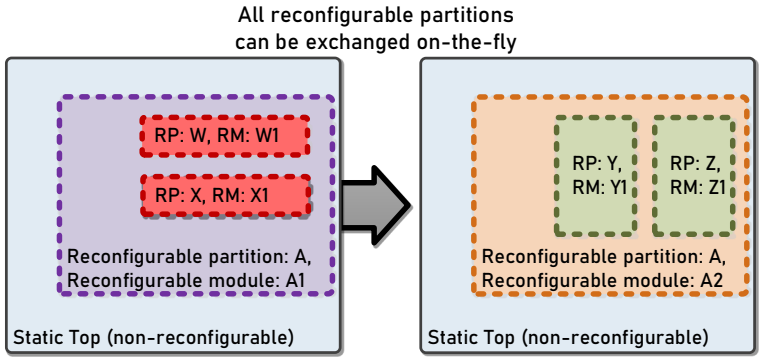

Figure 3.26: Nested Dynamic Function eXchange, as presented in [21]

Nested Dynamic Function eXchange [21] involves the ability to dynamically and selectively modify specific hierarchicaly organized sections or layers of an FPGA design, while preserving the overall functionality and integrity of the device. This hierarchical approach allows for multiple levels of customization and reconfigurations. It enables flexibility in loading different configurations, even on-the-fly, and managing user applications independently.

This workflow is an iterative process where, at each stage, the design is analyzed from the perspective of a single static and a single dynamic region within it. Bitstreams are generated for each stage as if it were a traditional partial reconfiguration process at a depth of one level. However, after this step, the compilation flow enters the inner dynamic region, treating it as a new static region. At this level, the boundary between the initial static and dynamic regions effectively disappears. This process continually shifts the context of static and dynamic boundaries, facilitated by the Vivado tools.

Ultimately, users are left with an initial static region bitstream and additional dynamic region bitstreams that encompass all possible hierarchical configurations. It's essential to note that all dynamic regions at every hierarchical level are treated in the same manner and must be adequately floorplanned.

An illustration of nested reconfiguration within Coyote is presented in Figure 3.27. In this example, the initial configuration comprises four vFPGAs, a shared RDMA stack, and an HBM stack. Over time, the high-performance RDMA stack is replaced with a TCP/IP stack, potentially as a part of a transition from data to control plane operations. Simultaneously, as two applications complete their tasks, vFPGA resizing takes place, leaving two active. Finally, a high-resource-usage application with high priority needs the entire FPGA dedicated to it, prompting a reconfiguration that removes all services and allocates as much fabric as possible to this single

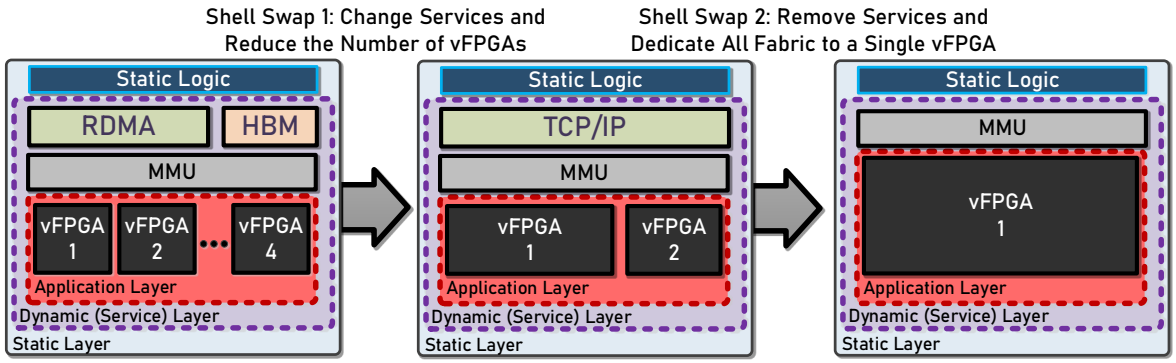

Figure 3.27: Nested Dynamic Function eXchange applied within Coyote

application. Such complex operations are made feasible by employing a hierarchical approach.

It is also advised by the vendor to go only up to three levels of dynamic reconfiguration (this is in line with Coyote's two level deep hierarchical approach). Looking ahead, there's potential to explore the addition of a third layer within the user logic. This layer would enable even finer-grained reconfiguration, fully dedicated to end-users.

The nested DFX implementation in Vivado relies on a project-based flow that exclusively utilizes scripting and is different than the traditional GUI-based DFX flow. This flow generates multiple projects and subsequently links their synthesized designs together. Although the nested DFX flow is fully functional, it predominantly relies on just two additional Tcl commands, specifically designed for partitioning and recombining dynamic regions and their cells. Beyond these commands, there is limited built-in support, and Vivado users are responsible for implementing the remainder of the design build flow. These commands are provided in the Table 3.5.

To understand better the implementation of hierarchical dynamic reconfiguration in Coyote, we need to delve into the broader build system used.

Table 3.5: Nested reconfiguration Tcl commands

Command	Arguments	Explanation
`pr_subdivide`	*parent and sub cells*	Break up parent cell into sub cells (RP one level down)
`pr_recombine`	*parent cell*	Restore parent cell (bring RP one level up)

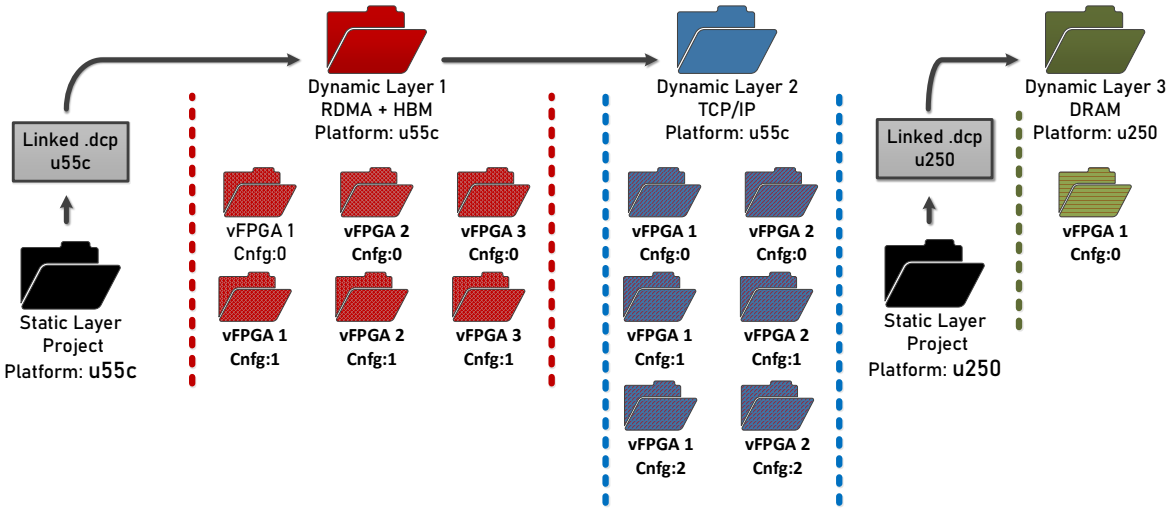

Figure 3.28: Example of Coyote project structure.

3.3.5 Build Flow

Conventional build infrastructure in general-purpose architectures is another feature that is notably absent in these heterogeneous systems, and in FPGA-oriented development in general. This infrastructure typically encompasses well-established compilers, code builders, assemblers, linkers, and loaders, providing developers with a significantly higher level of productivity. FPGA tools, in contrast, are usually vendor-specific, and their toolchains often come with rudimentary build systems, lacking any kind of modularity. Thus it can be challenging to integrate these devices into higher-level build environments.

To address this issue, we made the decision to employ a combination of tools and languages that are not strictly oriented solely for hardware development. One of these tools is CMake, which is commonly known for its use in software development to manage the build process of software projects, including support for compilation, incremental builds, linking, and packaging source code to produce executables. We drew inspiration from previous work that showcased the effective utilization of this toolchain for hardware development, with a particular focus on high-level synthesis. [90].

In addition to this, we attempt to leverage higher-level scripting languages, such as Python, wherever feasible, especially for code generation tasks. Regrettably, a complete transition to this scripting language was not possible as the actual compilation of bitstreams must be executed

within vendor-specific tools, which adhere to specific language requirements and predominantly rely on Tcl. Consequently, a blend of these two scripting languages has proven to be an efficient combination for orchestrating the overall build flow.

Project Creation

The "project" flow within Vivado offers the capability to synthesize individual projects independently and subsequently merge them into a single overarching project after the synthesis step. This feature is pivotal when working with dynamic reconfiguration, as it enables these distinct projects to serve as different reconfiguration regions. In Coyote, this means that multiple projects will be created as an initial step:

1. **Static layer**: In this project, only the static layer and its associated circuitry will be included. The dynamic layer will be treated as a black box and won't contain any logic.
2. **Dynamic layer (the user shell)**: This project will encompass the circuitry responsible for managing the operation of the dynamic layer, as well as any other selected services within the shell. Importantly, it won't contain any user applications; they will also be treated as black boxes.
3. **vFPGA projects**: These are multiple projects, each representing a separate application. The number of these projects depends on two factors: 1) the number of vFPGAs within the dynamic layer, and 2) the number of different configurations of these vFPGAs within the overarching shell.

An example of this setup is illustrated in Figure 3.28. Here, two different dynamic layers are created for the Alveo-u55c device, and one for the Alveo-u250 device. Each of these dynamic layers comes with its own configuration and a specific set of vFPGAs.

In this specific example, each dynamic layer is associated with a set of services and a certain number of vFPGAs, 3 and 2, respectively. Additionally, the first dynamic layer has 2 different configurations for its vFPGAs, while the second one has 3 different configurations. These configurations represent different applications synthesized separately. You can think of them as a small library of accelerators. It's worth noting that all these vFPGAs also need to be floorplanned, and this can be done either manually within the respective projects or by providing external constraints. The same approach applies to other devices, with the only difference being, the linking is performed with a different static design checkpoint.

For each hardware device, there exists only one static layer project. Typically, the static project is not created from scratch; instead, a design checkpoint file is provided, containing a netlist of the synthesized static layer. The dynamic layers can then be linked to this synthesized checkpoint.

Listing 3.5: The example CMake configuration for three dynamic layers.

```
find_package(CoyoteHW REQUIRED)
...
# Dynamic layer 1
set(FDEV_NAME "u55c")
set(N_REGIONS 3)
set(N_CONFIG 2)
set(EN_RDMA 1)
set(EN_MEM 1)
...
# Dynamic layer 2
set(FDEV_NAME "u55c")
set(N_REGIONS 2)
set(N_CONFIG 3)
set(EN_TCP 1)
...
# Dynamic layer 3
set(FDEV_NAME "u250")
set(N_REGIONS 1)
set(EN_MEM 1)
...
create_hw()
```

Listing 3.6: Insertion of kernels into vFPGAs of the second dynamic layer.

```
load_apps (
    VFPGA_0_CNFG_0 "some_path_to_kernels/hyper_log_log/"
    VFPGA_1_CNFG_0 "some_path_to_kernels/hyper_log_log/"
    VFPGA_0_CNFG_1 "some_path_to_kernels/count_min"
    VFPGA_1_CNFG_1 "some_path_to_kernels/count_min"
    VFPGA_0_CNFG_2 "some_path_to_kernels/agms"
    VFPGA_1_CNFG_2 "some_path_to_kernels/agms"
)
```

An example of a CMake configuration file for Coyote, used to set up the described scenario is shown in the Listing 3.5. After the final call to `create_hw`, which will validate the chosen configuration, all accompanying projects can be created by executing a simple make command.

An additional function `load_apps` is provided within CMake. This function simplifies the integration of user applications into all of the generated vFPGAs. It selects the compute kernels (both HLS and RTL) from specified directories, compiles them if necessary and inserts them into the vFPGA wrappers created by the project generation scripts. This streamlined process reduces the time required for incorporating applications and provides a degree of automation,

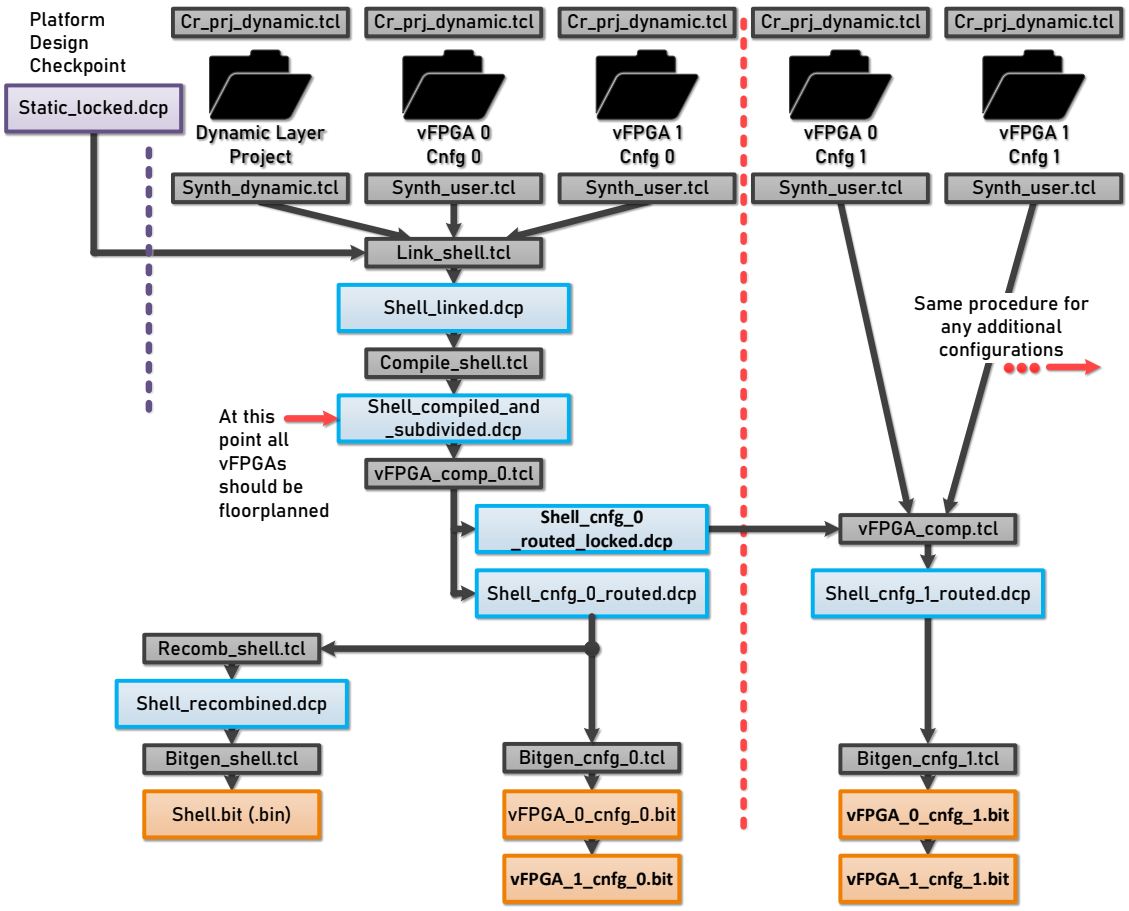

Figure 3.29: Coyote compilation flow.

particularly for users who may not be well-versed in low-level FPGA languages such as Verilog or VHDL. These languages are necessary for connecting all the interfaces in vFPGAs.

After all the necessary logic has been inserted, the synthesis can be initiated. The synthesis is performed in parallel, with each project being synthesized separately. This approach tends to make the synthesis process somewhat faster than the subsequent steps, where the netlists are mapped to the actual FPGA fabric (placement, routing) and which can be very time consuming.

After synthesis, the linking occurs. connecting synthesized regions to the design checkpoint with a pre-synthesized static layer. This step results in a single linked design. Following this, a traditional compilation flow begins, including placement, physical optimizations, and routing.

Upon completing this compilation, another design checkpoint is generated, which contains the compiled shell. At this stage, the `pr_subdivide` command (shown in Table 3.5) is applied. This results in the creation of multiple different designs, equal to the number of vFPGA configura-

tions. In these designs, the only dynamic regions are the vFPGAs, and the upper dynamic layer becomes static.

Another compilation step is initiated at this stage, with a focus on the vFPGAs. First, the initial configuration (*Cnfg 0*) is compiled. This process creates an initial design that locks in the vFPGAs. Therefore, it is always advisable to place the most complex circuits in this initial configuration. The checkpoint containing the locked design serves as the basis for the compilation of other configurations.

Each of these vFPGA-layer compilations produces a separate checkpoint, which are in turn used to generate all the vFPGA bitstreams. Finally, the initial configuration is recombined (with a call to `pr_recombine`) to generate the overarching dynamic layer bitstream, marking the completion of the full compilation flow. The whole process is shown with an example in Figure 3.29.

3.3.6 Shell Floorplanning and Resource Usage

Floorplanning in FPGAs involves defining the precise physical positions of logic elements within the FPGA's programmable fabric. It is a valuable technique for enhancing FPGA performance, routing efficiency, and overall resource utilization. This can be executed in a manual fashion, where designers personally determine the placements, or it can be automated using algorithms that optimize placement while adhering to specified user provided constraints. Automated floorplanning is seamlessly integrated into the toolchain during compilation and typically operates without user intervention. Nevertheless, it's essential to recognize that even though these automation is robust, it might not always yield optimal results since the tools often lack a comprehensive understanding of the higher-level design hierarchy.

When using partial reconfiguration, manual floorplanning becomes mandatory for separating the target reconfigurable region from other active areas in the FPGA. Partition pins are inserted automatically to enforce this isolation physically. These pins "lock" existing crossing interfaces, maintaining a clear distinction between the dynamic and static portions of the design. Careful consideration is necessary as these crossings significantly limit routing flexibility in the vicinity of the floorplanned regions, which can lead to substantial routing congestion.

Thus careful approach to floorplanning of all dynamic regions is necessary. In Coyote, the first dynamic region is the dynamic layer (user shell) containing all of the management logic, services and the end user applications. The floorplanning of this region is provided by default and remains fixed. This is due to the static layer being fixed as discussed in previous sections.

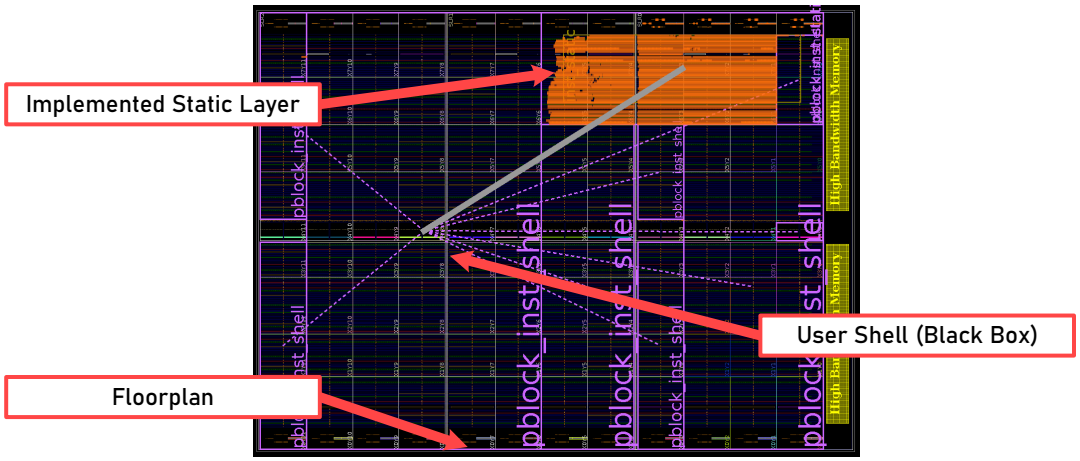

Figure 3.30: Locked static layer and floorplanning for Alveo-u280 (u55c) board

The static layer is minimal and occupies minimal amount of FPGA fabric. The rest is dedicated to the actual system which runs on top (the dynamic layer itself).

Another consideration which needs to be taken into account is any potential I/O within the dynamic layer. All I/O ports in the FPGA require buffering to preserve the signal integrity and ensure proper logic propagation. These buffers are hardware primitives in the FPGA chip and are inserted automatically by the tools in the top level of the hierarchy. As the top level constitutes the static layer and these ports are now exposed directly to the dynamic layer, tool needs to be instructed to not insert these buffers. This is done through the use of special pragmas (`io_buffer_type = "none"`) within the top layer. These ports are then passed directly to the dynamic layer where manual insertion of explicit buffers is performed.

While this procedure may appear relatively uncomplicated, it demanded a dedicated effort to accurately understand the finer intricacies, pragmas, constraints, and the overall workflow. Much of this knowledge is not easily accessible, is often proprietary, and necessitates significant reverse engineering efforts. Once more, this example underscores the challenges associated with the development and build processes when working with these devices.

The Figure 3.30 displays the completed floorplan for the Alveo-u280 board. In this representation, the logic within the static layer is highlighted in orange, while the dynamic layer, depicted as a black box, is enclosed by purple pblocks. These pblocks house all the logic accessible to the dynamic layer. This floorplan minimizes the usage of space for the static region, ensuring compliance with device constraints and achieving the desired timing closure. Although further refinements in floorplanning are possible, they are unlikely to yield significant improvements.

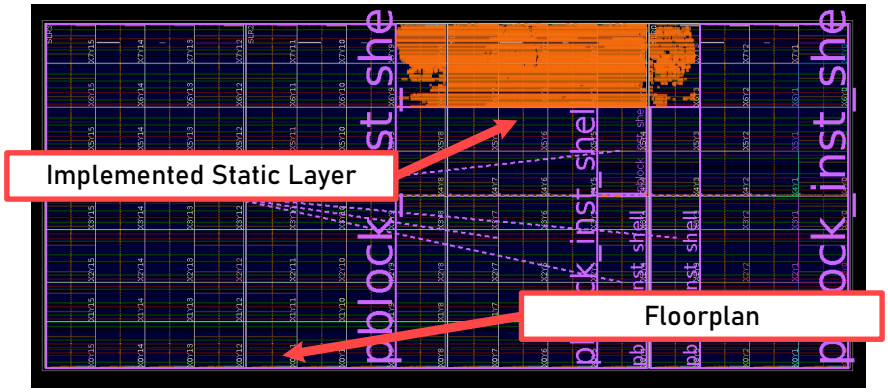

Figure 3.31: Locked static layer and floorplanning for Alveo-u250 board

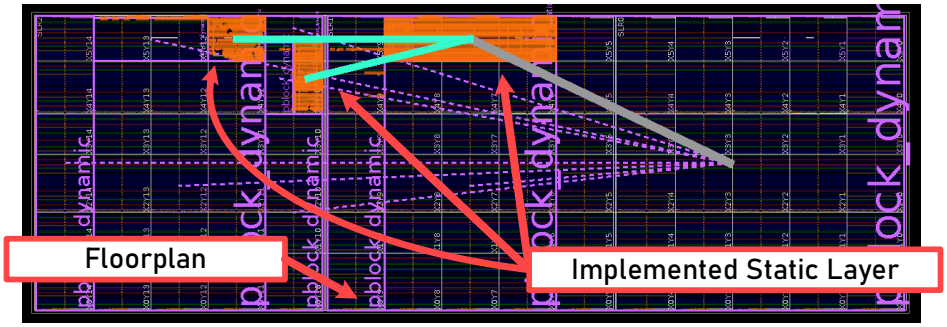

Figure 3.32: Locked static layer and floorplanning for Enzian board

It's worth noting that the same floorplan is applied to the u55c board as it contains the same chip. The design checkpoint, housing this locked static layer, is available in the official Coyote repository and is maintained by the system's developers. This consistent static layer and its floorplan is provided to all Coyote users that are targeting this particular board. This allows the static layer to be always "online" in the clusters, enabling the loading of user-provided dynamic layers, regardless of where they were built.

Comparable floorplan is done for the Alveo-u250 board (Figure 3.31). This device's chip includes an additional SLR, resulting in more FPGA fabric. This extension has somewhat simplified the floorplanning process and allows the static layer to occupy slightly less space proportionally, thereby providing more room for the dynamic layer and, ultimately, user applications.

Simultaneously, floorplanning is conducted for Enzian (Figure 3.32). Although Enzian is using FPGA chips with the same architecture as vcu118, it features a notably distinct connection to the host CPU. The ECI link in Enzian employs 24 lanes, connected through 6 groups of quad

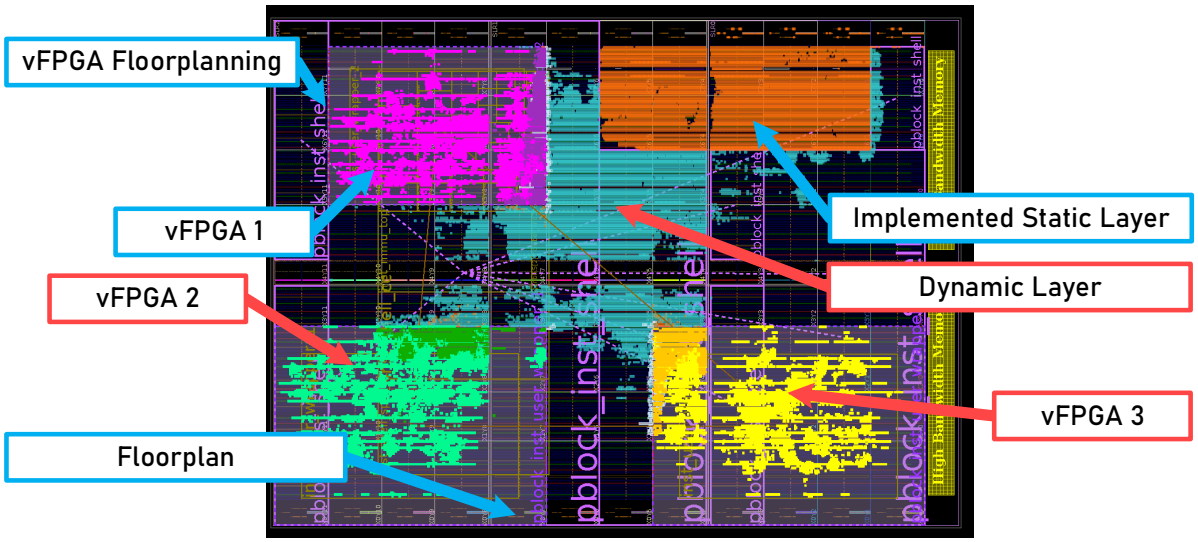

Figure 3.33: Floorplanned vFPGAs within the dynamic layer. Initial configuration of the sketch algorithms with loaded Hyper-Log-Log circuits (Alveo-u55c).

transceivers, in contrast to the 4 groups used in PCIe. This leads to a more extensive distribution of the static layer, spanning two SLRs, which does complicate routing somewhat.

The floorplanning of user applications in the vFPGAs is not done automatically within Coyote and this step is left to the end users. The primary reason for this approach is the complexity of covering all potential use cases and configurations that may arise. Moreover, the floorplanning process is highly dependent on the specific deployment scenarios, making it challenging to create all-encompassing abstractions for this feature.

The actual floorplanning process is relatively straightforward. Users need to define a rectangular area (known as a "pblock") within the FPGA fabric, designating the location for their logic. This constrained block serves as the boundary for the provided applications in Coyote. The key challenge here is to ensure that this design fits well within the overall system architecture and that the embedded logic meets the necessary resource and timing requirements.

An example of such floorplanning is illustrated in Figure 3.33 where three applications representing sketch algorithms (Hyper-Log-Log, Count-Min, Fast-AGMS) are each placed in three separate vFPGAs. These three algorithms exemplify a standard acceleration library that can be built in the Coyote framework. It's important to note that this floorplanning met all the necessary timing requirements, although it may not represent the optimal floorplanning effort by any means.

Table 3.6: Static and dynamic layer (3x Hyper-Log-Log) reosource consumption.

Hardware Platform	CLBs	LUTs	Registers	Block RAM
Alveo u50d	16 781 (15.4 %)	76 939 (8.82 %)	86 332 (4.95 %)	96 (7.14 %)
Alveo u200	15 627 (10.57 %)	77 852 (6.58 %)	84 815 (3.58 %)	96 (4.44 %)
Alveo u250	15 675 (7.26 %)	77 875 (4.51 %)	84 297 (2.44 %)	96 (3.57 %)
Alveo u280 (u55c)	15 016 (9.21 %)	77 741 (5.96 %)	84 378 (3.24 %)	96 (4.76 %)
vcu118	15 168 (10.26 %)	77 694 (6.57 %)	84 496 (3.57 %)	96 (4.44 %)
Enzian	9018 (6.1 %)	56 098 (4.74 %)	51 188 (2.16 %)	71.5 (3.31 %)
HLL x3 (User shell, u55c)	18 174 (11.17 %)	78 248 (6.02 %)	166 333 (6.39 %)	100 (4.97 %)
HLL x3 (vFPGAs, u55c)	16 348 (10.06 %)	62 898 (4.83 %)	96 480 (3.69 %)	195 (9.66 %)

Resource Usage

We will compare default floorplanned static layers on various platforms and the dynamic layer with sketch algorithms (Figure 3.33). The resource usage is shown in Table 3.6.

We can observe that the overall resource consumption of Coyote's static layer is less than 10% of the overall CLB usage in almost all cases (exception being a relatively small FPGA chip in Alveo-u50d). This leaves a significant amount of fabric available for the dynamic layer, which was the primary objective. This indicates that we can continue with the comparisons between the static layer and the conventional concept of a kernel. The usage of various resources is fairly consistent, with no single resource showing notably high usage.

We can also see the resource distribution for the user shell, which includes 3 Hyper-Log-Log operators. It consumes approximately 10% of the total resources. In this specific scenario, a significant portion of these resources is utilized by register slices and TLBs, which are both by default very generously provisioned to improve the overall timing and memory coverage. As they are parametrizable, their overhead can be decreased if needed.

This implies that the overall Coyote overhead for three concurrent applications employing traditional FPGA-side acceleration is approximately 20%, which is well within an acceptable range. This becomes even more apparent when we consider that deployed applications, in conjunction, utilize only up to 10% of the total resources. These sketch algorithms, which are involved in stream processing, represent common applications deployed on modern FPGAs and exhibit typical resource overheads. Once more, this emphasizes the significance of implementing a multi-tenant shell for modern FPGAs to avoid resource underutilization.

3.4 Summary

In this chapter, we explored the ”kernel” of Coyote, which offers the fundamental functionality required for higher system levels. We introduced platform interfaces that are built on top of the CPU-FPGA interconnect, demonstrating how standardizing these interfaces enhances system portability, as exemplified by the Enzian port. Additionally, we explored hierarchical dynamic reconfiguration, which serves as the primary source of flexibility in Coyote.

In the next chapter, we will explore the first dynamic layer, which offers crucial isolation and security features, as well as the various services provided by the system.

Chapter 4

Dynamic Layer: The User Shell

This chapter explores Coyote's dynamic layer, encompassing a variety of services available to the system's end users. These are highly adaptable and can be tailored to meet a diverse range of deployments. They are also modular, making it feasible to introduce additional extensions in the future, particularly well-suited for a versatile device such as an FPGA.

Staying within the realm of operating system comparisons, we can draw a parallel between the dynamic layer and an OS "shell" that handles functions such as memory, I/O, and process management. The central function of the dynamic layer is to implement these crucial system services, exposing them to the user applications within the system. We can draw another analogy with the hypervisor model frequently employed in Virtual Machine (VM) environments.

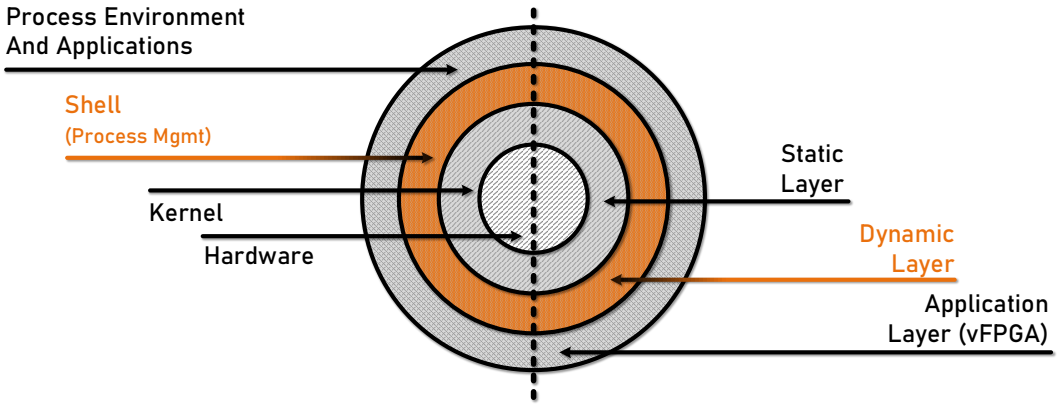

Figure 4.1: Dynamic Layer.

In the context of Coyote, the hypervisor is represented by the combination of the static and dynamic layers, providing vital functionality and granting access to essential hardware and services within the system. We lastly have the user applications and their corresponding trusted wrappers. The wrapper concept aligns well with the idea of a "virtual machine monitor", with one dedicated to each distinct VM (or vFPGA in Coyote), ensuring essential isolation within the system and providing secure access. The applications running within the trusted wrappers can be compared to conventional software executed in VMs.

This chapter will begin with a high-level overview of the dynamic layer and then proceed to explore the critical memory and networking services. This will also serve as a foundation for transitioning to the application layer where we will explore the Unified Logic Interfaces (ULI) and the ultimate software abstractions in Coyote.

4.1 The Architecture of the User Shell

The high-level overview of Coyote's hardware infrastructure is depicted in Figure 4.2. The dynamic layer is constructed on top of the static layer interfaces detailed in the previous chapter. As a result, this layer is platform-agnostic, maintaining uniformity across all devices running Coyote. This characteristic is pivotal in facilitating the incorporation of additional services within this layer while efficiently abstracting the inherent complexities found in the initial static layer and host to FPGA communication. Since all the I/O ports designated for these dynamic services are directly accessible within this layer, it falls upon the developers of these services to decide which I/O to employ and how they intend to utilize it. This offers an added degree of integration flexibility, as none of the I/O options within this layer are predetermined or fixed.

All the services within this layer are accessible to and shared among all the tenants. The management of these tenants, scheduling of the services, and safeguarding critical system components are all coordinated within this dynamic layer.

4.1.1 Expansion of the Static Layer Interfaces

The interfaces that bridge the static and dynamic layers are designed to be as minimalistic as possible, without sacrificing functionality. The reason for this is the very high cost of crossing between static and dynamic regions in FPGA fabric. Additionally, keeping these interfaces light allows us to use less space for the static layer. Nonetheless, as we transition into the

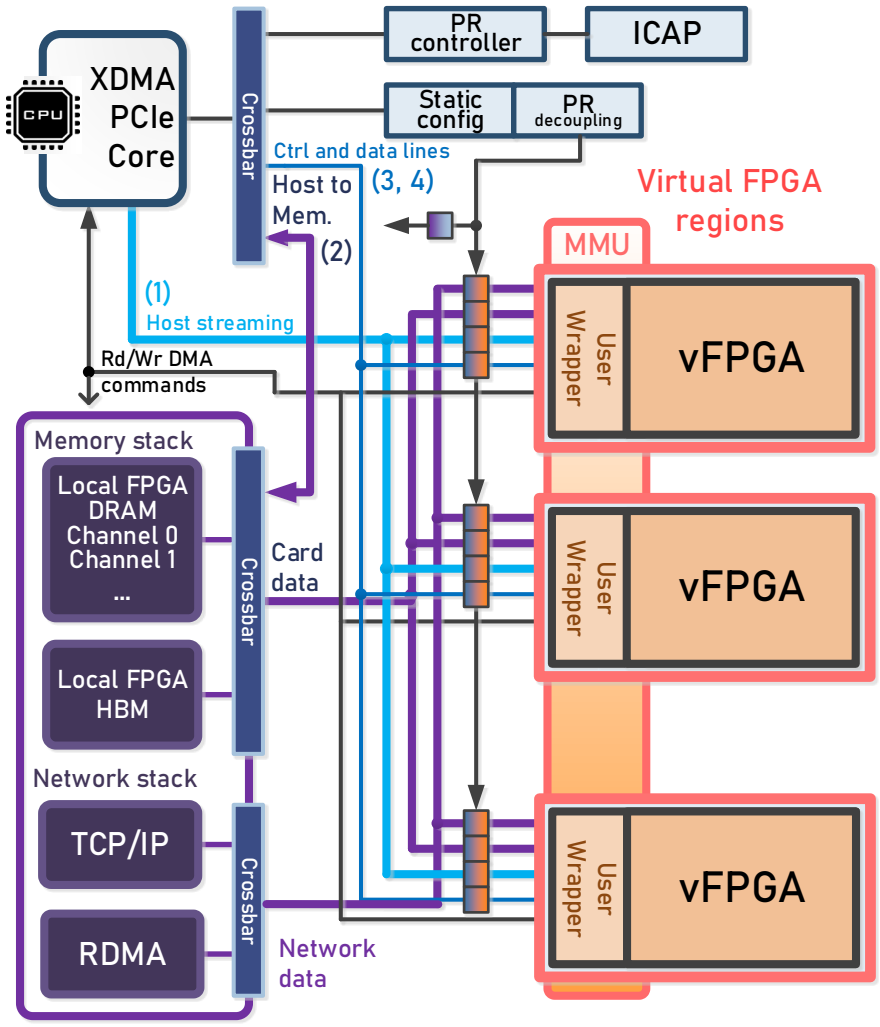

Figure 4.2: High level overwiew of Coyote's hardware infrastructure

dynamic layer, these interfaces must expand and adapt to meet the demands of the multi-tenant environment.

In any digital circuit, the propagation of clock signals is of utmost importance. Dynamic layer is initially supplied by a single clock that is sourced from the static region. This clock is primarily designed to optimize interconnect performance, and serves as the primary system clock source. However, a single clock source is not universally suitable for all the circuits and modules.

In various scenarios, it proves highly advantageous to have different sections of the circuits operating at distinct clock frequencies. For example, network stacks can already fully saturate

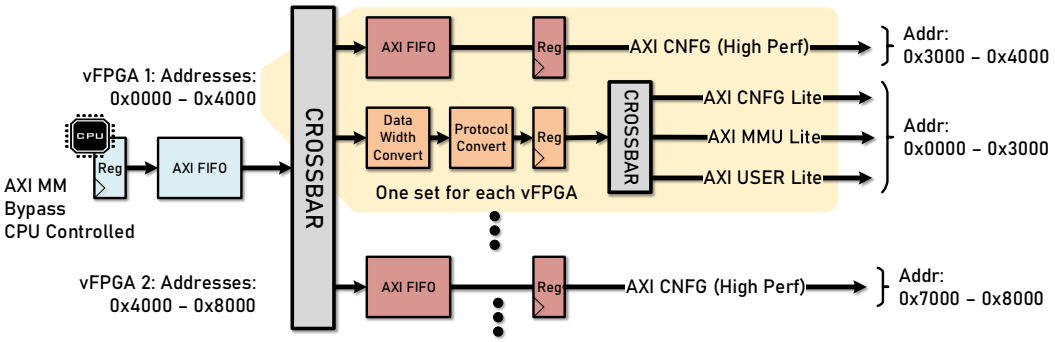

Figure 4.3: Control arbitration tree. Each vFPGA control interface is memory-mapped with independent address space.

100 Gbps links at 250 MHz, and running them at a higher frequency (although possibly reducing latency slightly) will probably not significantly impact overall throughput during bulk network transfers. Simultaneously, memory components may require a slightly higher clock frequency to tap into their full bandwidth potential – typical DRAM operations run at 300 MHz, highlighting the clock discrepancy between different modules.

This clock frequency discrepancy can extend to user provided circuits, which might be compute-bound with extensive pipelining that can effectively handle much higher clock rates (up to 500 MHz). In this case to efficiently exploit performance from all parts of the system multiple clock sources need to be available. To address these different clock requirements, it becomes essential to generate a clock tree that can accommodate these modules, each operating at its specific clock frequency. The clock tree in the dynamic layer is generated by the PLL (Phase Locked Loop) located in the dynamic region. Most of the output frequencies of the PLL are modifiable and can be adapted to specific application needs. The PLL generates the top level system clock for the dynamic layer, separate memory and network clocks, and an additional user defined clock for end user applications.

Now, let's turn our attention to the control plane. The control plane enters the dynamic region via a full AXI4 memory-mapped bus. This particular "bypass" bus is accessible through a dedicated PCIe BAR, typically sized at a few megabytes. This BAR serves as the conduit for controlling every aspect of Coyote from the dynamic layer onwards. It is connected to a series of crossbars that facilitate its distribution throughout the system. This interface holds significant importance, as it provides the main control available to the host CPU. It serves both the driver (for operations such as migrations), the software API (for operations that initiate and control data movement), and even the end users as an "all-purpose" direct memory-mapped interface to interract with

their deployed hardware applications. Direct memory mapping of this interface ensures efficient operation without kernel overheads. Asynchronous control in the reverse direction (from FPGA to CPU) is established through interrupts. Each vFPGA has a dedicated physical interrupt line, whereas the interrupts within a single vFPGA are consolidated or multiplexed into a single one.

4.1.2 Fair-Sharing and Packetization

The remaining interfaces to the static layer consist of the data movement descriptors and data lines, serving both the streaming and migration channels. The data within the streaming channel must be accessible to every tenant, necessitating the sharing of this bus. The sharing is essential since the potential bandwidth in the FPGA far surpasses that of the CPU-FPGA communication link. However, this sharing must be impartial as all tenants within the system have the same priority. Failing to distribute this bandwidth fairly could potentially lead to certain tenants being starved of resources, which would be contrary to the core multitasking objective of Coyote.

Furthermore, each tenant has the capability to read or write substantial amounts of data and initiate transfers that may span hundreds of megabytes. It is imperative to ensure that these transfers are consistently completed without impacting the operation of other tenants. It's worth noting at this point that the logic running within each tenant is fundamentally untrusted and could potentially be adversarial. In the chapter dedicated to the application layer, we will delve into the strategies for ensuring smooth operation and securing the system against any potential issues stemming from the tenants (logic within vFPGAs). Our primary focus now is on achieving fair-sharing, assuming a cooperative and non-adversarial behavior from the applications.

To ensure that even distribution of the resources is effectively achieved, we implement a packetization strategy for all data requests. This enables us to finely manage all transfers within the system by controlling all outstanding transactions. The default packet size in Coyote is set at 4 kB (parameter, can easily be changed), which proves more than sufficient to saturate all links, both local and remote. End users retain the flexibility to submit requests of varying sizes, yet each of these requests will be automatically divided into packet-sized chunks at the appropriate layer. The system handles packets seamlessly, without requiring any direct intervention from end users.

The packetization is complemented by interleaving (Figure 4.4), which is employed to distribute constrained resources, such as the limited bandwidth of the host or network links. Interleaving is achieved through a conventional round-robin arbitration scheme to guarantee equitable resource allocation within the system. The key enabler for interleaving is the in-order handling of all data

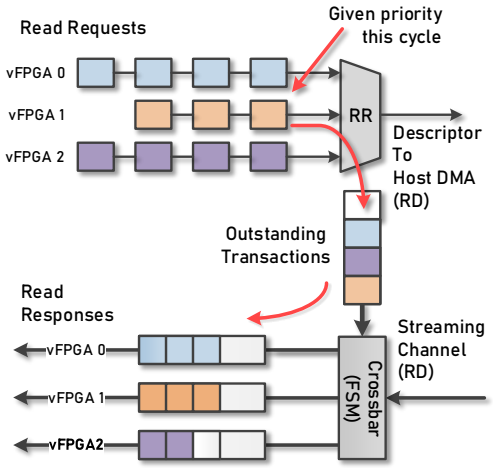

Figure 4.4: Interleaving vFPGA requests destined for host memory.

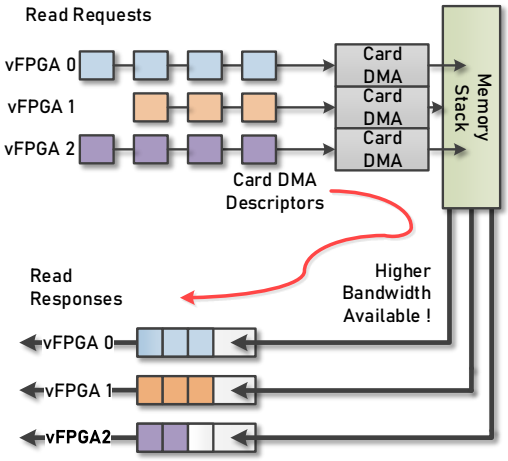

Figure 4.5: Parallel vFPGA requests destined for card memory.

packets. This combination of packetization and interleaving is applicable to most of the data movements throughout the system, regardless of their origin, whether local or remote.

Interleaving is not implemented for the interfaces accessing the FPGA-side memory stack (Figure 4.5). This stems from the fact that there's no necessity to interleave these requests, primarily because of the significantly higher bandwidth provided by the local FPGA memory. This superior bandwidth permits us to efficiently parallelize interfaces from each vFPGA, ensuring that each of them can fully utilize the available bandwidth. The task of managing these requests is subsequently delegated to the internal components of the memory stack, which are responsible for overseeing accesses to memory resources.

4.1.3 Evaluation

In this evaluation, we aim to assess the capability of Coyote's multitasking approach in providing fair-sharing across multiple vFPGAs concurrently. This is particularly crucial in cloud deployments where stable and predictable distribution of resources is a fundamental requirement.

Our test scenario involves running up to eight concurrent applications, in eight different vFPGAs. We've chosen to employ AES (Advanced Encryption Standard) encryption cores for these tests. AES is a widely used symmetric-key encryption algorithm that supports key lengths of 128, 192, or 256 bits for data encryption and decryption (128-bit keys used in the tests).

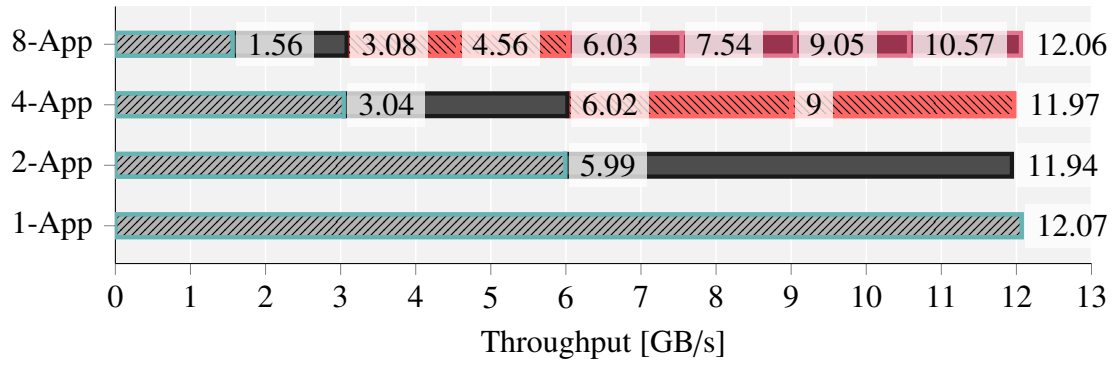

Figure 4.6: AES bandwidth distribution (Counter mode - *memory-bound*).

The selection of AES is deliberate for several reasons. Firstly, it serves as a clear-cut example of an application where FPGAs excel, and is often found deployed on these devices in real-world scenarios [66]. Secondly, AES comes in various flavors and, significantly, offers both parallelizable and non-parallelizable modes. This duality allows us to examine Coyote's behavior with memory-bound and compute-bound cores, respectively.

AES has a variety of modes of operation. Some of the modes are parallelizable, allowing the circuit to pipeline the input words and thus achieve higher performance and better utilization of the hardware pipelines. The most notable parallelizable mode is ECB, but comes with poor security guarantees. The alternative often used is the Counter mode, which provides a parallelizable circuit that can achieve an initialization interval of 1, while providing much higher security gurantees. We will thus use this mode of operation for our compute-bound test that will stress the bandwidth of the host link.

In the second test, we'll use the widely adopted CBC (Cipher Block Chaining) mode (known for its higher levels of security [195]). It's important to note that CBC mode is not parallelizable, which means it is compute-bound and will not fully utilize the available host link's bandwidth.

We assess the per-application round-trip throughput by conducting tests on a series of batched large transfers, each consisting of 8MB of data sent 1000 times (Alveo-u55c). Up to 8 applications concurrently initiate these data requests, which involve sending plain data to the FPGA user logic, performing pipeline computations, and simultaneously transmitting the computed results back to memory. The transfers are batched from the host, and we retrieve completion notifications by polling the host memory using writeback functionality.

The throughput results for the compute-bound scenario are shown in Figure 4.6, while the results for the memory-bound test (CBC mode) are depicted in Figure 4.7. Examining the throughput

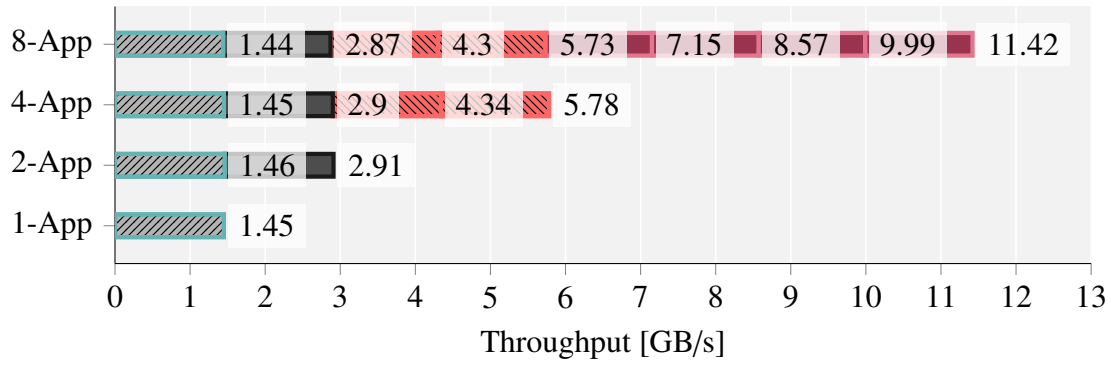

Figure 4.7: AES bandwidth distribution (CBC mode - *compute-bound*).

graph for the compute-bound AES, we notice that the performance scales as anticipated across applications, provided that all the virtual FPGAs can be accommodated within the FPGA.

In the memory-bound case multiple AES cores are contending for PCIe bandwidth, which is consistently saturated across all instances due to the highly pipelined nature of this AES implementation. Each of these AES cores has the potential to maintain a theoretical throughput of 19.2 GB/s. It is noteworthy that, the throughput of an AES instance exhibits an inverse relationship with the number of applications in the system, indicating that the limited PCIe bandwidth is divided among them. Additionally, the throughput across applications remains consistent in all instances, indicating that the resource sharing is balanced. To sum it up, these findings affirm our objectives of fairly distributing the limited FPGA bandwidth among multiple tenants.

It is interesting to look at the resource usage of encryption cores at this point (shown in Table 4.1). In this case, we successfully accommodated 8 high-performance parallel encryption cores within a single dynamic layer on the Alveo-u55c board. Nevertheless, this represents a practical limit, as pushing further would introduce excessive congestion and make achieving reasonable timing closure quite challenging (even though extensive optimization efforts were not undertaken in this context). Hence, there are constraints on the scalability of applications (vFPGAs) in modern FPGAs.

Table 4.1: AES (x8) resource usage (Alveo-u55c).

Configuration	CLBs	LUTs	Registers	Block RAM
User shell + Static layer	117 569 (72.15 %)	598 711 (45.92 %)	544 277 (20.87 %)	488 (24.21 %)
AES cores	758 041 (46.52 %)	383 546 (29.44 %)	136 957 (5.28 %)	136 (6.72 %)

4.2 Shared Virtual Memory

In this section, we will dive into the FPGA-side services in Coyote. Much like services in an operating system, these are integral components of Coyote, and they hold a pivotal role in effectively managing FPGA-side resources. They importantly also provide abstractions, enabling FPGA application developers to interact with critical system elements like network and memory without the need to concern themselves with hardware intricacies. This greatly enhances the productivity of FPGA application development.

These services are thus an essential part of Coyote, serving as a foundational layer that manages hardware resources, abstracts complexities, and facilitates the efficient multitasking of concurrent virtual FPGAs.

We will start exploring the memory services and their structural organization. We will explore the virtualization layer and the holistic memory management system employed in Coyote. This encompasses a comprehensive layer that necessitates support across the entire spectrum, from the management of physical memory at the hardware level to the specific software libraries within the host OS kernel. The main purpose of these layers is to provide a *virtual memory abstraction* akin to the one typically found on conventional machines.

Virtual memory is a crucial abstraction for serveral reasons:

- Primarily, it provides security by segregating processes within their individual, private address spaces. This segregation serves as a barrier, preventing any malicious process from affecting other processes or tampering with their memory. For a system like Coyote, which aims to establish a multi-tenant environment, this abstraction stands as a critical security feature.

- It creates the illusion of a more extensive memory capacity than the actual physical resources. Consequently, it frees the system from the restrictions of the underlying physical memory. In scenarios involving devices like FPGAs, which incorporate diverse memory types, this abstraction surpasses the physical constraints of these memories, freeing users of the need to be concerned with the specifics of these memories and their quantity.

- Conceals the complexity of dealing with physical addresses. In multi-tenant environments, virtual memory efficiently shields end-users from the details associated with managing

physical addresses. Dealing with virtual address ranges is considerably more straightforward than grappling with shared physical addresses in a multi-tenant environment.

- Virtual memory affords the flexibility of dynamic memory allocation and the possibility of memory sharing. This flexibility paves the way for optimizing memory allocation patterns. This feature enables us to investigate diverse allocation models and access patterns, and it is the very foundation for some of the memory optimizations taking place within Coyote. Furthermore, it removes the necessity for fixed address compilations, simplifying the tasks of compilers and, ultimately, streamlining the end user programming model.

4.2.1 Background

Regrettably, FPGAs rarely incorporate virtual memory and commonly depend on the use of physical addresses within user FPGA applications to access resources [208]. Given that the majority of FPGA shells do not offer memory virtualization support [122, 62, 212, 210, 86] in the FPGA, the responsibility of handling address translations for accessing host-side resources falls entirely upon the host device driver. This approach to the general memory model comes with significant limitations:

- **Extensive OS involvement and CPU overheads**: The heavy reliance on the host device driver to supervise and actively participate in data transfers between the CPU and the FPGA leads to substantial OS kernel involvement and large overheads. As a result, a considerable number of valuable CPU cycles is consumed.

- **Lack of seamless virtual address pointer exchange**: The inability to exchange virtual address pointers between software running in user space and FPGA-side applications is a significant limitation. This capability is particularly critical for applications dealing with complex pointer-based data structures. It substantially simplifies the programming model for both CPU-side and FPGA-side applications. In a model without memory virtualization, such seamless interactions are infeasible.

- **Absence of FPGA-Side "mastering" capability**: The absence of the ability for FPGA applications to independently manage data transactions originating from the FPGA side represents another important constraint. This capacity grants FPGA-based applications the authority to manage operations more comprehensively, providing them with greater freedom to explore diverse processing techniques and abstractions. This level of control

is especially beneficial in situations where extensive host CPU involvement, such as in graph-based pointer-chasing, would otherwise be necessary, or where certain operations, like RDMA, would be entirely unachievable.

In rare instances where virtual memory is implemented on FPGAs, it tends to fall into two categories: one targeted at System-on-Chip embedded platforms [199] and the other for large-scale proprietary systems [188], both of which have their limitations.

In the case of embedded architectures, they often lack high-performance interconnects, powerful CPUs, and other significant abstractions like multi-tenancy. This is primarily due to the limited space available on SoC FPGAs, which constrains the inclusion of such features.

On the other hand, in the latter scenario, where virtual memory is designed for data centers, exemplified by OpenCAPI [188], it relies on dedicated, fully hardened blocks. While this approach offers reliability and efficiency, it comes at the cost of flexibility, with only a limited set of features available. Additionally, other abstractions are frequently absent, and there is limited room for adding extra functionalities due to the hardware constraints.

The closest FPGA shell to explore virtual memory akin to Coyote is Optimus [146]. Like Coyote, it provides multi-tenancy and the ability to seamlessly exchange pointers between user space and FPGA. However, the similarities end there. Similar to OpenCAPI, Optimus relies on a hardened IOMMU (it employs a standard CPU-attached IOMMU). This setup provides very limited flexibility for implementing additional abstractions, as the host IOMMU is fixed.

Coyote implements virtual memory by creating a dedicated memory management unit within the FPGA fabric. This unit uses software-loaded Translation Lookaside Buffers (TLBs) to manage address translations. What sets Coyote apart to any other shell out there, is that this effectively decouples the virtual FPGAs not only from the host CPU but also, crucially, from other services within the FPGA (Figure 4.2).

This isolation is the cornerstone of system security, protecting the entire setup from the applications running in the virtual FPGAs. This approach allows us to regard these applications as untrusted entities while still delivering a comprehensive set of features and services. As a result, we can provide multi-tenancy, virtual memory (for both host CPU and FPGA side memory accesses), and offer high level networking abstractions like TCP/IP and RDMA.

4.2.2 Physical Memory Management

To begin explaining memory virtualization in Coyote, we must start at the physical layer. The versatility of FPGAs is notably exemplified by their I/O capabilities. FPGAs can accommodate a wide range of different memory types, including traditional UDIMMs, RDIMMs, LRDIMMs, and even novel High Bandwidth Memories (HBMs) with numerous memory channels for high throughput data transfers. However, this diversity poses a challenge in terms of usability, as integrating and adapting systems for each of these memory types can be quite cumbersome.

This is the primary objective of physical memory abstraction in Coyote: to conceal the intricacies of various memory types and offer a uniform interface to user applications, regardless of the specific memory attached. Furthermore, memory accesses should adhere to the same streaming interface employed consistently throughout the Coyote system. This entails using descriptors to specify the virtual address and length of the data transfer, followed by the provision of data through dedicated data streams.

The questions that arise are how to ensure the security and protection of this memory, how to efficiently manage dynamic allocation among virtual FPGAs, how to optimize access from numerous tenants, and how to establish virtualization and integration with the device driver and the host operating system.

The solution is to use the memory management unit in the FPGA fabric. By placing this memory behind the MMU and routing all accesses through the internal TLBs, we gain the capability to manage virtual-to-physical address translations according to our own requirements. Because these TLBs are software-loaded, we have the flexibility to handle dynamic allocation from the host operating system, enabling us to explore various allocation strategies and even allowing us to share memory locations between tenants. Through the implementation of different allocation schemes and effective mapping management, we can also optimize concurrent accesses from the virtual FPGAs, ensuring efficient use of the memory.

4.2.2.1 Striping

How does this virtualization layer assist us in efficiently optimizing memory accesses from the virtual FPGAs? The quantity of these virtual FPGAs can vary significantly, depending on the specifics of each shell deployment. This necessitates optimizing the system in a way that is agnostic to the number of virtual FPGAs and aims to maximize bandwidth for all users, while equally prioritizing fair resource sharing among them.

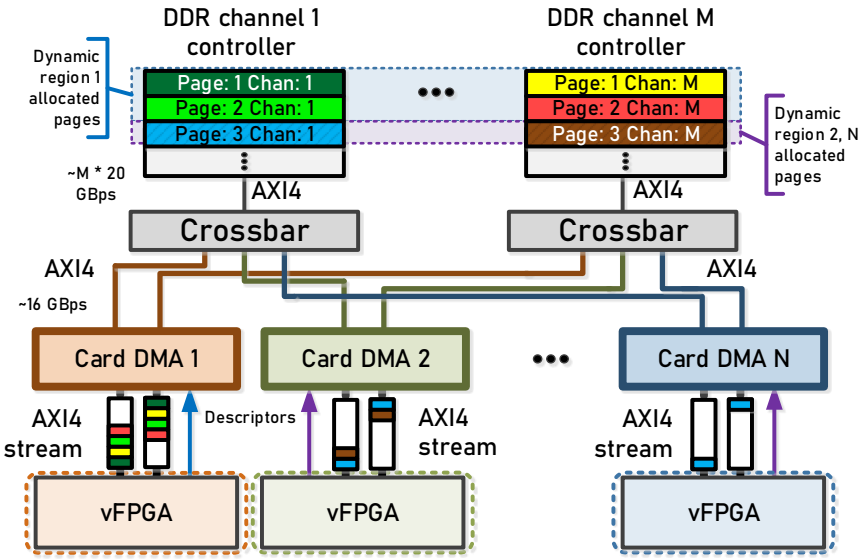

Figure 4.8: Dynamic allocation and striping access pattern

This optimization is achieved by implementing a striping access pattern that leverages the dynamic allocation capabilities within Coyote. It effectively allocates user memory in pages across all available memory channels, regardless of their number. For example, if a user requests a 4 MB buffer and there are 2 DRAM channels available, the allocated buffer will be divided into 2 MB chunks, each allocated to one of the two channels. The addresses within these channels are then interleaved. For instance, if a user makes an access for 128 bytes (equivalent to two channel data widths, representing two bus transactions), the lower 64 bytes (size of the cache line) are stored in one channel, while the higher 64 bytes are stored in the other channel. This interleaved arrangement of data words across these channels optimizes bandwidth because it enables simultaneous access to all channels.

Furthermore, this optimization extends to scenarios where multiple users access the memory, regardless of the number of users. Since the accesses are parallel and span across all channels, the achieved throughput remains at its peak, ensuring optimal data transfer rates (Figure 4.8).

However, this optimization comes with a trade-off. The interconnect responsible for propagating accesses from each virtual FPGA to all of the memory channels can be quite costly. This is because the memory controllers for each of these channels are not hardened and are instead instantiated in the FPGA fabric. As a result, this interconnect needs to reach all of the controllers, which in many cases can span across different SLRs. This additional logic and routing complex-

ity can lead to large congestion and ultimately timing violations (especially at SLR crossings).

This is one of the reasons why, even though Coyote supports 4 DRAM channels, they are rarely utilized on devices like Alveo-u250. The congestion they generate within the fabric often creates more problems than it solves and leaves little routing resources for other circuitry. Consequently, on these devices, it's common practice to enable only two DRAM channels, which has been found to be a feasible configuration, ultimately providing a well-routable system.

The absence of a dedicated hardened interconnect and hardened memory controllers in FPGAs is a drawback. However, new architectures, like new Versal chips [32], are beginning to emerge that incorporate both of these elements, along with some other hardened circuits within the chip. These advancements allows techniques like the striping we use here to be implemented with significantly reduced penalties.

4.2.2.2 High Bandwidth Memory Organization

Modern FPGAs integrated with high bandwidth memory represent a notable advancement in the realm of reconfigurable hardware and memory technology. The primary advantage of high bandwidth memory lies in its wide interface and stacked memory dies, which result in high data transfer rates. Additionally, it offers benefits in terms of reduced latency and power consumption.

In numerous modern systems, the FPGA and HBM share the same package, creating a multi-chip module. This physical proximity allows for shorter and more efficient connections. It also enables the use of dedicated hardened memory controllers. Unlike DRAM connections that must accommodate a wide range of chips, HBM has a fixed and static controller that can be implemented within dedicated circuits. Consequently, FPGAs equipped with HBM typically provide a number of full AXI interfaces that can be directly accessed without the need for additional control circuitry.

The architecture of HBM in HBM-equipped FPGAs like Alveo-u50d, u55c, or u280 is illustrated in Figure 4.9. HBM typically consists of 2 banks, each housing 16 pseudo channels. Each pseudo memory channel is 256 MB in size (512 MB on u55c). Each of these channels is equipped with a dedicated AXI interface. Between the interface and the channel, there is a switching interconnect that allows every AXI channel to access the entire HBM range. The highest performance is still achieved when accessing the channel physically closest to the corresponding AXI channel [202].

It's essential to highlight that this interconnect differs from traditional DRAM-equipped FPGAs where it used up the FPGA fabric. In this case, the interconnect is preexisting and fully hardened,

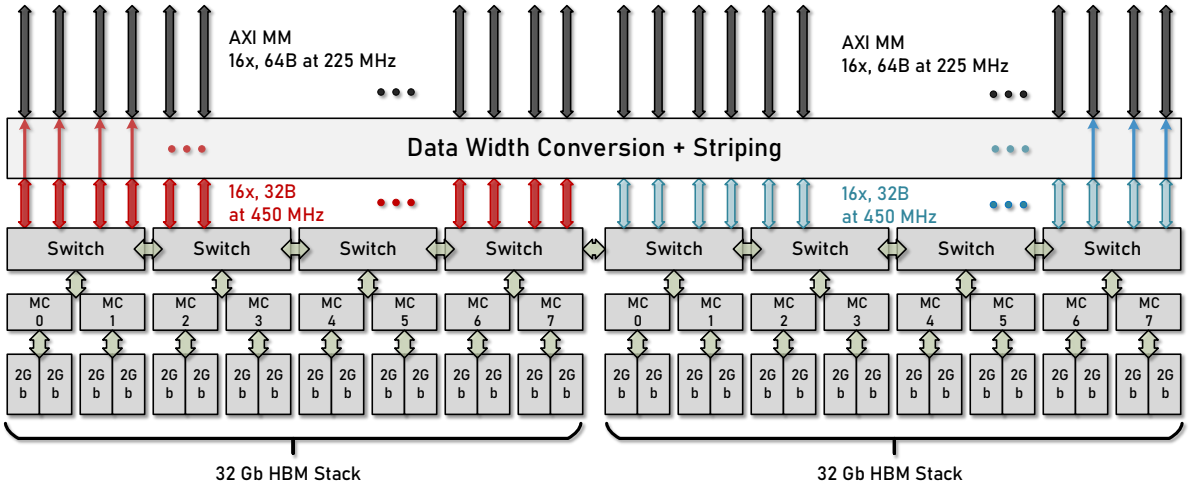

Figure 4.9: First configuration of HBM: Data width conversion

eliminating the need for additional FPGA resources. The capability to access every pseudo memory channel from any AXI interface is well-suited to Coyote's striping access pattern.

While HBM provides a significant number of channels, it was primarily designed to work effectively with GPUs and high-performance ASICs. The highest achievable throughput with practical FPGA applications is typically much lower than the maximum throughput that HBM can theoretically provide (which can exceed 400GB). This limitation primarily arises from the FPGA fabric's inability to efficiently accommodate a large number of parallel high-performance AXI buses and keep up with these extremely high data transfer speeds. Using 32 AXI channels at high frequencies and simultaneously deploying 32 applications capable of fully utilizing this performance within the FPGA is rarely feasible, can lead to substantial congestion and, consequently, major timing violations. These challenges are particularly pronounced in the part of the chip responsible for connecting the FPGA to the HBM.

As a result, applications using HBM in FPGAs tend to operate at lower frequencies or with fewer channels. In Coyote, users have the flexibility to choose the number of channels per vFPGA as they please. However, it's essential to recognize that in real-world usage scenarios, the achievable throughput is considerably smaller than the theoretical maximum due to these practical limitations.

Each AXI interface exposed from the HBM is 32-bytes wide. As we've established, standard bus size in Coyote is cache-line sized (64-bytes). Thus, an adapatation of the HBM's AXI channels to Coyote's card DMA AXI interfaces is needed.

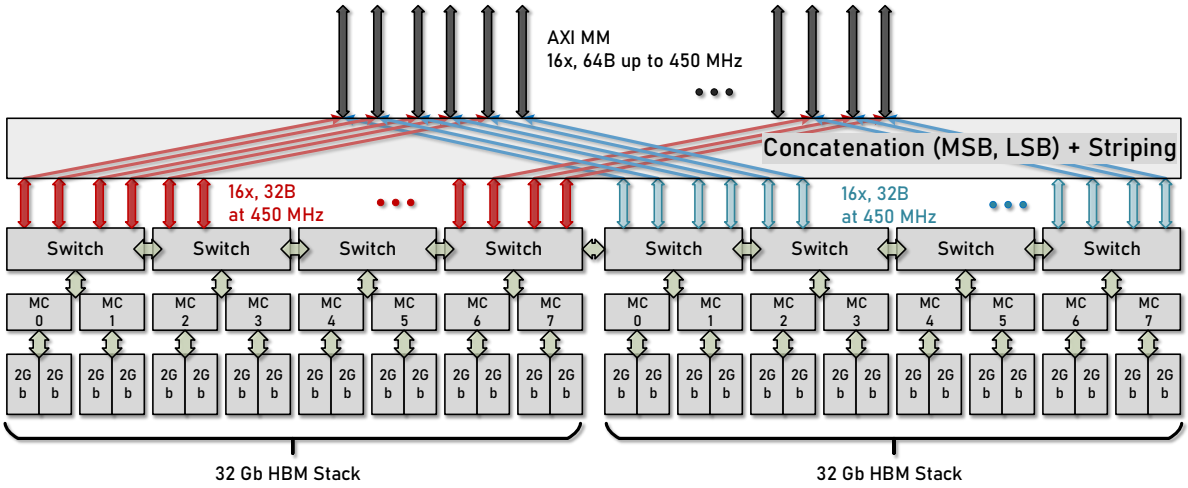

Figure 4.10: Second configuration of HBM: Concatenation of HBM banks

In the first approach, we go for a straightforward data width conversion, as depicted in Figure 4.9. The HBM channels operate at a very high frequency of 450MHz, which is typically lowered through explicit clock crossings when running any practical applications. We follow this approach at the system level by implementing a combination of clock crossing and data width conversion. The interface width is converted from 32 bytes at 450 MHz to 64 bytes running at 225 MHz (or at any higher system clock used at that point). Importantly, the maximum throughput is preserved:

$$Max\ thr. = [\texttt{at hbm}]\ 32B \cdot 450MHz = [\texttt{at vFPGA}]\ 64B \cdot 225(+)MHz = 14.4GB/s$$

Each vFPGA channel can thus provide the maximum throughput of a single pseudo channel, provided that the memory clock is equal to or higher than 225 MHz. This throughput is more than sufficient to accommodate all other system components (e.g., network, host interconnect). Additional concurrent channels can be instantiated to further enhance performance.

However, in certain cases, users may be running highly pipelined applications with very short critical paths that demand extremely high frequencies. These applications, for instance, might only be capable of utilizing a single memory channel. As shown, the maximum throughput achievable through this single channel in the previous configuration of the HBM is approximately 14 GB/s. However, if these applications are running at 400 MHz, the potential throughput is considerably higher ($\approx$ 25GB/s).

To accommodate such specific scenarios, Coyote offers an alternative HBM configuration. In

this setup, instead of creating the AXI channel on the Coyote side through data width conversion, we choose to concatenate two pseudo channels together. As a result, the 64-byte read or writes from Coyote are divided into two pseudo HBM channels in different HBM banks. This configuration, shown in Figure 4.10, allows us to effectively utilize a significantly higher bandwidth through a single vFPGA AXI interface. The maximum achievable throughput in this case is attained when running circuits at the maximum HBM frequency:

$$Max\ thr. = \texttt{[at hbm]}\ 2 \cdot 32B \cdot 450MHz = \texttt{[at vFPGA]}\ 64B \cdot 450MHz = 28.8GB/s$$

In this case, it's possible to achieve twice the performance through the same single channel as before. It's important to highlight here that these are various configurations provided in Coyote, and users have the flexibility to choose the one that best suits their specific deployment. It's also noteworthy that all these changes and different memory configurations (DRAM, HBM) do not affect end users at all. The interface remains consistent, creating a standard execution environment that promotes portability and ease of use. This consistency allows for the exploration of different configurations without impacting the end users and their applications.

4.2.3 Performance Evaluation of the Memory Stack

In this section we will measure the performance of the memory stack in a few different configurations. We will examine the throughput of both DRAM and HBM with multiple tenants accessing these memories. The tests for the DRAM memory are conducted on Alveo-u250 boards, while the tests with HBM memories are conducted on Alveo-u55c boards.

Unlike the constraints and difficulties posed by softcore DRAM controllers in the case of Alveo u250, where it is challenging to achieve meaningful results with 4 DRAM channels, HBM offers greater scalability and thus more memory channels can be utilized. In this test, four vFPGA tenants will perform accesses to the FPGA-side memory. The test will cover a variety of transfer sizes. These accesses are concurrent and are initiated in batches from the CPU (host initiated). Consequently, there will be some overhead, particularly for smaller transfer sizes. Since HBM can be scaled with vFPGAs, each of these four vFPGAs will have its dedicated HBM interface. On the u250 board, only two DRAM channels are enabled, necessitating some multiplexing and arbitration by the memory stack.

The throughput results are depicted in Figure 4.11. These results illustrate that at this level (with 4 vFPGAs), HBM scales effectively, enabling us to fully capitalize on the parallel channels

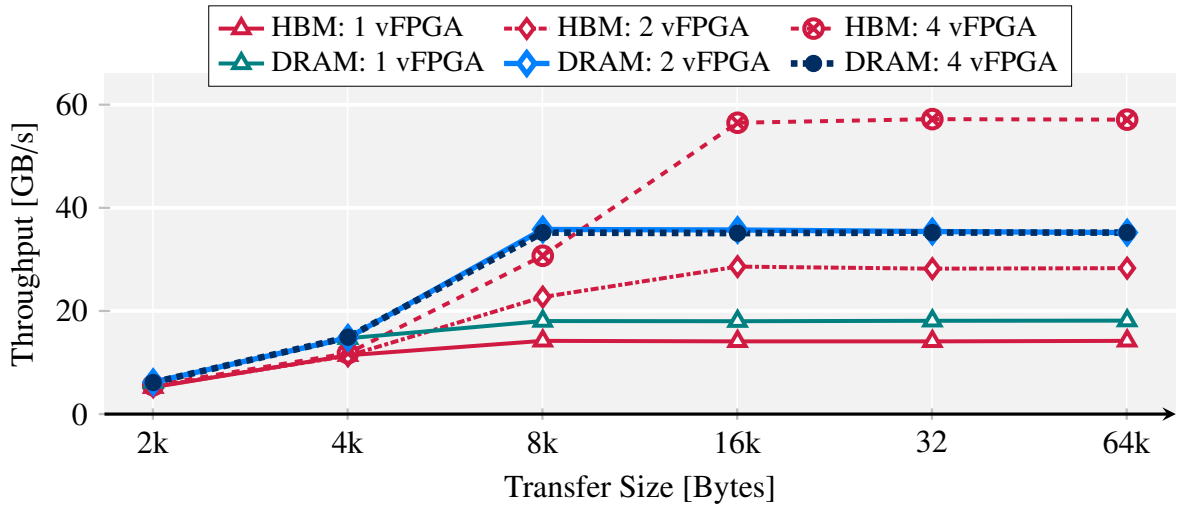

Figure 4.11: Memory stack throughput.

within it. However, as only two DRAM channels are active on the u250, these become the limiting factor when more than two tenants concurrently access the memory. It's worth noting that DRAM exhibits a slightly higher per-channel throughput, but this disparity can be mitigated by enabling Coyote's second HBM configuration.

In practice, we managed to scale HBM up to 8 concurrent channels without significant problems. However, when exceeding this point, congestion levels increase significantly, necessitating adaptations such as reducing system clocks, which ultimately lead to performance degradation.

It's also interesting to note the latency of these memories when accesses are made directly from the vFPGA applications. We compare the averaged latencies to the base reported latencies that could be achieved in HBM and DRAM [202]. The results are shown in Table 4.2.

Table 4.2: DRAM and HBM latencies in Coyote.

Configuration	DRAM latency [ns]	HBM latency [ns]
Coyote	182.8	220.4
Base performance	73.3	106.7

It's apparent that Coyote introduces some overhead to the peak performance achievable with these memories. This overhead arises from the virtualization layer, DMA initiations, and the added arbitration logic required to implement striped access patterns. However, the overhead is relatively modest, especially considering the level of abstraction it provides.

Lastly, we will report a resource usage for the HBM and DRAM configurations (encompassing the full FPGA design), utilizing the Alveo-u280 board. We selected this board because it incorporates both HBM and two DRAM channels, making it the most suitable platform for an equitable comparison.

Table 4.3: Resource usage of shells with different memory configurations (4 vFGPAs, Alveo-u280).

Configuration	CLBs	LUTs	Registers	Block RAM
DRAM - 2 chan.	48 037 (29.48 %)	216 244 (16.59 %)	382 603 (14.67 %)	329.5 (16.34 %)
HBM - 4 chan.	51 962 (31.89 %)	238 772 (18.32 %)	407 130 (15.61 %)	304 (15.08 %)
HBM - 8 chan.	78 900 (48.42 %)	350 866 (26.91 %)	611 732 (23.46 %)	392 (19.44 %)

4.2.4 Memory Management Unit

Now that we covered the physical management of the memory within the FPGA, we can focus on the actual memory management unit that orchestrates all memory accesses within the system.

The Memory Management Unit (MMU) in Coyote plays a central role as the primary protection layer in the system. It serves as the main module responsible for virtualizing user applications (vFPGAs). Inside the MMU, each vFPGA is equipped with its dedicated software-loaded Translation Lookaside Buffer (TLB). These vFPGAs are essentially virtual devices within Coyote, utilizing the same virtual address space as the host CPU and the corresponding processes.

In conventional operating systems, each user process maintains its distinct virtual address space. The operating system takes responsibility for managing these address mappings using page tables stored in the host memory. These mappings translate virtual addresses into corresponding physical addresses. This translation process is executed by the CPU's dedicated memory management unit, allowing processes to interact with physical memory without requiring knowledge of the precise physical locations. The isolation between the processes and physical memory is critical, serving as a cornerstone for ensuring memory protection and access control. It guarantees that one process remains completely isolated from and incapable of interfering with or accessing the memory of another process, thereby preserving system integrity and security.

However, performing a translation by accessing the host memory for every request would introduce unacceptable overhead. To mitigate this, a faster translation method is employed by using translation lookaside buffers. TLBs serve as hardware caches closely integrated with the memory management unit to expedite the translation process. In essence, a TLB functions as a

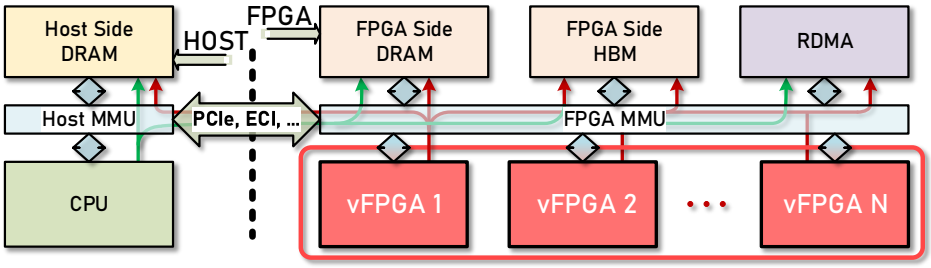

Figure 4.12: Virtualization layer and decoupling of user applications in the system.

compact memory with high random-access performance that stores a select set of recently translated virtual-to-physical address mappings. When the TLB contains the necessary translation (a TLB hit), the MMU can swiftly retrieve the associated physical address from the TLB, resulting in a much quicker memory access. However, if the TLB lacks the required translation (a TLB miss), the MMU must execute a complete translation. This process takes more time because it necessitates accessing the page table stored in the host memory to locate the mapping. In such cases, a full page-table-walk within the host MMU becomes necessary, introducing additional latency and overhead.

A page table walk stands as a fundamental operation within the MMU. Its primary purpose is to find the physical page for a set of virtual addresses that missed in the TLB. The MMU initiates this process by accessing the page tables (typically multi-level pointer chasing) to discover the mapping for the given virtual address. Once this mapping is identified, it is cached within the Translation Lookaside Buffer (TLB), and the operation is reattempted. This second attempt results in a TLB hit, allowing the operation to proceed.

The core concept in Coyote is to integrate FPGAs into this process, eliminating the need for a simplified, bare-metal-like physical memory model commonly found in most modern FPGA setups. To achieve this integration, we expand the host TLBs to the FPGA side, effectively creating an IOMMU within the FPGA fabric.

While high-end CPUs often include IOMMUs to manage and secure memory access for external devices, Coyote takes this a step further. In Coyote, the IOMMU is significantly more potent:

- Firstly, being integrated into the FPGA fabric, it offers unparalleled flexibility, making it adaptable to any given scenario. In Coyote's context, this adaptability enables us to create parallel TLBs capable of simultaneously managing the system's multi-tenancy. Addition-

ally, these TLBs can be configured with arbitrary sizes. As they are instantiated in the dynamic layer, they can also be resized on-the-fly as required during shell reloading.

- Secondly, it provides protection for the services operating within Coyote. The memory and network components in Coyote incorporate high-level abstractions, such as RDMA. Ensuring the security of these services from applications within the FPGA would not be possible using a standard host CPU IOMMU.

The MMU layer serves a dual role, functioning both as an arbiter and a coordinator. It manages access requests originating from end-user applications, directing them to the appropriate modules within the system. These requests may target diverse locations, including memory on the host CPU side, the FPGA side, or even remote memory accessed via RDMA.

Each user request includes details about the intended memory target. From the moment these requests reach the MMU, it becomes the MMU's responsibility to validate the request's mappings, verify user permissions for memory access, and subsequently orchestrate the request until it is successfully completed.

4.2.4.1 Translation Lookaside Buffers

We will now look into the organization of the TLBs that manage all local node accesses. The TLBs within Coyote are shown in Figure 4.13. They are fully parametrizable thus the size of the TLB (number of rows) and the set-associativity (number of columns) can be chosen arbitrarily between different dynamic layers (user shells).

The TLBs are implemented using on-chip memory, specifically Block RAM, which is purpose-built for high-performance data manipulation in FPGAs. This type of memory is particularly valuable for swiftly handling random-access operations with an exceptionally low latency (just a single cycle). In this respect, it shares similarities with traditional cache memory.

However, like caches, Block RAM also faces certain limitations. Its high-performance characteristics come at the cost of reduced density, and it's typically available in limited quantities, often in the order of megabytes in modern FPGAs. Furthermore, it is distributed across the entire chip, which imposes additional restrictions on its utilization.

These TLBs are responsible for managing traditional OS page mappings, similar to the way a conventional CPU's TLB operates. However, in this context, the cost of a TLB miss is significantly higher because the overhead associated with a page fault from an external device is much more substantial than that of a page fault within the host CPU's MMU.

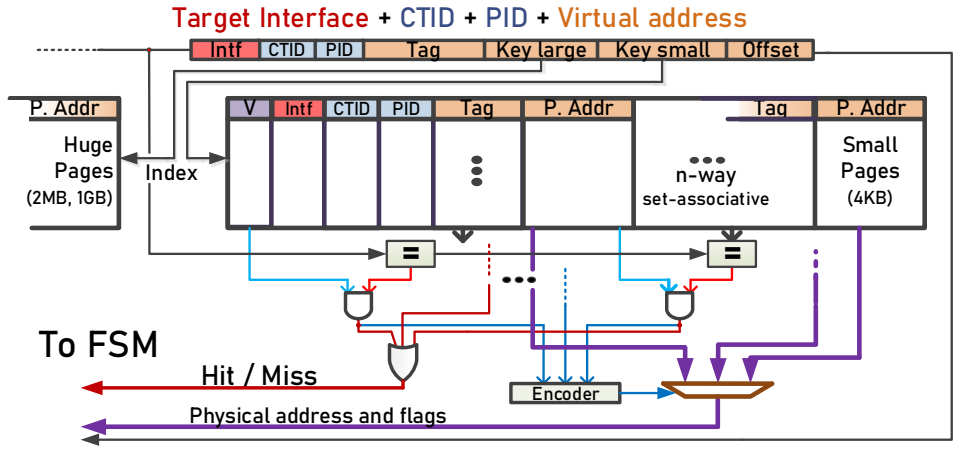

Figure 4.13: Translation Lookaside Buffers in Coyote.

To mitigate this issue, it's crucial to ensure that these TLBs offer comprehensive coverage, minimizing the occurrence of page faults. Considering a standard 4 kB OS page size, it's evident that a conventional approach involving replicated 4 kB TLBs on the FPGA would not provide sufficient coverage. For instance, even a relatively large 1 MB TLB would only be able to accommodate less than 400 MBs worth of host OS pages. Given today's big-data-driven world, it's evident that this level of coverage is inadequate.

Hugepages

To address this challenge, we turn to the solution of using *hugepages*. These large-sized memory pages prove to be an ideal remedy for this particular use case. They are instrumental in enhancing system performance by minimizing the overhead linked to managing smaller memory pages, and their utilization is common in operating systems to fine-tune system efficiency, especially when there's a substantial demand for memory, such as in virtualization scenarios.

In the context of Coyote, the most significant advantage of hugepages is the notable reduction in TLB misses and their ability to offer much broader memory coverage compared to traditional 4 kB pages. To take advantage of these large pages, we provide parallel TLBs with parallel lookups, enabling simultaneous operation. These parallel TLBs have variable page sizes and thus users have the flexibility to choose their preferred hugepage size to meet their specific requirements. To illustrate the benefit of hugepages, when 1 GB hugepages are activated alongside 64 kB TLBs, the potential coverage extends to approximately 5 terabytes, a well-sufficient capacity for target Coyote applications.

TLB Fields

Each TLB entry contains additional fields, beyond the address information of the mapping. The first of these is the *target interface* field, which holds crucial information about the location of the actual physical data associated with the mapping. This field can indicate whether the data resides in the host memory or the FPGA-side memories. If the current value of this field doesn't align with the one in the incoming request (even if the tags match), it will result in a page fault. This page fault is then handled by the driver in a specific manner (a process we will explore in one of the next sections). Ultimately, this page fault will initiate a migration process to relocate the data to its required destination.

The second set of fields comprises identifier fields, specifically the *hpid* and *ctid*. The hpid stands for the standard process ID from the host OS, and it serves the purpose of managing TLB entry invalidations. As Coyote relies on CPU MMU invalidation notifiers to keep TLBs up to date, the *hpid* field is used to match the mappings which need to be cleared when invalidation commands arrive.

The *ctid*, on the other hand, represents the *Coyote Thread ID*. This thread ID introduces an additional multitasking dimension, unique to Coyote. Beyond running multiple vFPGAs in parallel, Coyote allows for the execution of multiple host threads within a single vFPGA, which can correspond to different host OS processes. This *ctid* field is used to enforce protection and identification at this level. A deeper exploration of Coyote threads will be covered when we dive into the application layer 5.3.1.

Evictions

In TLBs, the eviction algorithm determines which of the set-associative entries are to be removed when a new one arrives and all slots are filled. While LRU (Least Recently Used) is an ideal choice, it's less common in hardware due to its complexity, resource requirements, and potential latency overheads.

For a more hardware-friendly solution, Coyote employs the NRU (Not Recently Used) algorithm for evictions. This algorithm assesses pages based on when they were last accessed. To implement this, we use two bits to indicate the page's "rank." First bit is the reference bit, which is set when the page is read. Second bit is the modify bit, which is set when the page is altered through a write operation. Additionally, a dedicated timer circulates through the TLB, periodically resetting the reference bits for all set-associative entries within TLB rows. The number of bits required to implement this algorithm is not extensive. It amounts to twice the set-associativity of the TLB multiplied by the number of TLB rows, resulting in a small overhead.

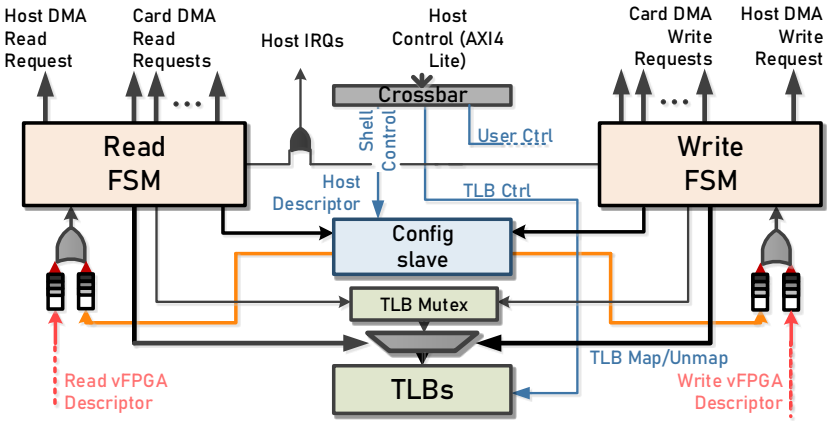

Figure 4.14: Read and Write Engines

4.2.4.2 Invalidations and Page Faults

The MMU employs dedicated engines responsible for managing all data requests. These engines are designed as finite state machines, governing the flow of data requests. The MMU is equipped with a set of these engines for each of the vFPGAs, enabling parallel operation. Furthermore, both the read and write data paths operate concurrently, enhancing parallelism in handling requests. This concurrency is crucial because, in typical acceleration scenarios, both read and write operations are issued simultaneously. In the standard operation mode, these engines handle data requests by initially verifying the TLB for the existence of accurate mappings. If the mappings are detected, the engines then proceed to partition these requests into pages (comprising both normal and huge pages) and subsequently transmit translated entries to the respective DMA units. The synchronization between the read and write engines is achieved through the hardware mutex located in front of the TLB.

The state machine responsible for managing these requests within the engines during regular operation is depicted in Figure 4.15. It's evident that in the absence of page faults, the requests will be seamlessly passed on to the DMA, with minimal intervention from the read/write engines. Consequently, this approach incurs minimal overhead in the overall request flow.

A critical operation that can disrupt this "normal" request handling is host invalidations. These invalidations refer to actions initiated by the host kernel, signaling to the MMU that specific virtual address range, belonging to certain processes, are about to be invalidated. Therefore, the MMU must update the values within the TLBs to reflect these operations.

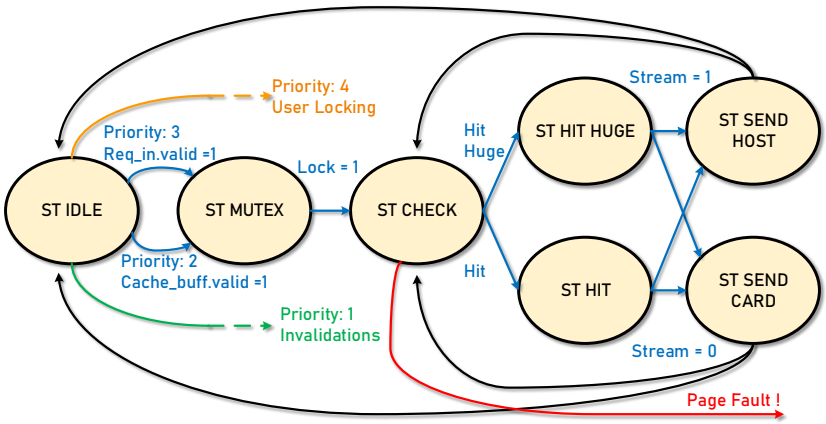

Figure 4.15: State Transitions During Normal Operation.

Invalidations

The original reasons behind these invalidations can vary. They might occur due to routine memory deallocation of vFPGA memory, or could be a result of host process memory being swapped out to disk, among other possibilities. How this operation is carried out and the general handling of memory between the host and the FPGA is a fundamental distinction in Coyote compared to standard FPGA integration into host kernels.

Typically, external accelerators pin the host memory while it's in use by the accelerators, imposing additional constraints on the operating system and its memory management. In contrast, Coyote offeres a different approach. Instead, the accelerator relies entirely on MMU notifiers to keep its TLBs up to date, thus avoiding constraints on the host kernel and allowing for a more transparent memory management. It's important to note that for legacy reasons, Coyote does offer traditional page pinning methods for maintaining TLBs.

Since these invalidations are tightly integrated with the host operating system and directly impact the host MMU, it is imperative to respond to them in an organized manner. Otherwise, user applications might attempt to access physical memory that has been released or reassigned to another process, potentially leading to system corruption.

Handling these operations is facilitated through a specialized API within the Linux kernel. Linux has introduced the `mmu_invalidate_notifier` function, precisely designed for similar purposes for which we employ it. To utilize these notifiers, a kernel module registers a set of callbacks with the kernel for a particular host process. These callbacks are then invoked whenever a mapping in the page table is invalidated for a specific address range.

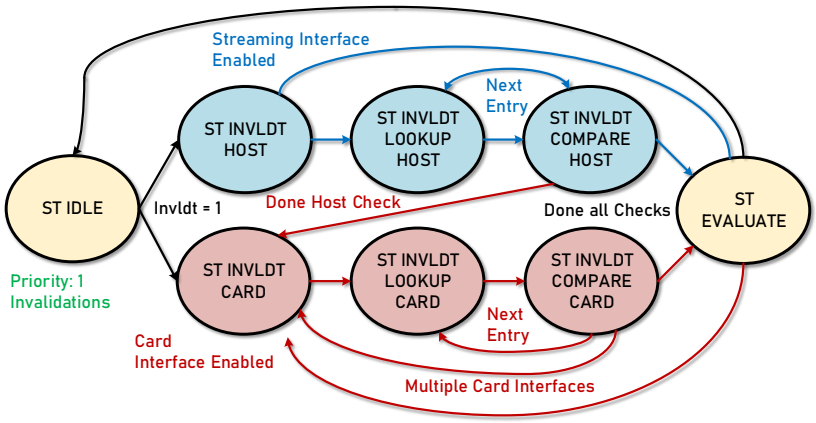

Figure 4.16: State Transitions During Invalidations.

Once these callbacks are triggered, we issue these invalidations to the FPGA's read/write engines, informing them about the impending invalidation. These invalidation messages include the host process ID and the virtual address range to be invalidated. The code of these invalidaitons is shown in the Listing 4.1.

Listing 4.1: Invalidation callback from the MMU notifier.

```
if (mmu_notifier_range_blockable(range))
    mutex_lock(&d->mmu_lock);
else if (!mutex_trylock(&d->mmu_lock))
    return false;

mmu_interval_set_seq(interval_sub, cur_seq);

// TLB unmap
first = start >> PAGE_SHIFT; last = end >> PAGE_SHIFT;
n_pages = last - first;
tlb_unmap_hmm(d, start << PAGE_SHIFT, n_pages, hpid, huge);

// Wait for invalidation ACK
wait_event_interruptible(d->waitqueue_invldt,
    atomic_read(&d->wait_invldt) == FLAG_SET);
atomic_set(&d->wait_invldt, FLAG_CLR);

mutex_unlock(&d->mmu_lock);
return true;
```

However, an additional concern arises during these invalidations. The kernel may request a specific invalidation, but at that moment, we might have in-flight transactions within the address range scheduled for invalidation. This implies that our vFPGAs are actively accessing this data, and the kernel cannot proceed with invalidation until these operations have been fully executed.

Since these operations might endure for some time, the callback function must have the capability to enter a sleep state. Relying on polling would be prohibitively expensive in this context. Consequently, the kernel needs to be alerted once it's permissible to carry out updates to its own page tables. In Coyote, we accomplish this through an invalidation acknowledgment, which is issued via a dedicated interrupt line. The callback function, therefore, goes into a sleep state until the acknowledgment is received, signaling to the kernel that the invalidation process is complete.

A trivial approach here to determine the existence of any outstanding transactions would be to wait until all transactions within a particular vFPGA are completed. However, as previously mentioned, Coyote supports multiple processes per single vFPGA, and this approach could potentially introduce substantial overhead. To address this, we implement dedicated circular buffers that store the ranges of all outstanding transactions in the system for each of the parallel data channels, including one streaming channel and a number of potential parallel FPGA side memory channels.

Duing invalidatations the state machine enters states (Figure 4.16) that examine these circular buffers for potential colliding addresses that are currently in-flight. If such collisions are detected, the engine will remain in these states until the outstanding transactions are resolved. One of the key benefits of this approach is that it ensures that all other in-flight transactions, associated with different Coyote threads and processes, are not required to finish their respective operations. Consequently, invalidations can be executed with reduced overhead, as only the specific pages being invalidated are examined.

Page Faults

Finally, in situations where the mapping, returned from the TLB, is incorrect, such as when the ID fields do not match, the addresses are inaccurate, or the data is located in a different location, a page fault is triggered. This situation initiates a separate page fault handling mechanism within the read/write engines. A specific section of the state machine (Figure 4.17) is tasked with managing these page faults, guaranteeing that the details about the specific page fault are relayed to the host for resolution.

In the event of a page fault, the engines will notify the host driver via a dedicated interrupt line. The driver acknowledges this interrupt and retrieves all the essential information regarding the page fault (virtual address, length, etc.). After handling the page fault and installing new mappings, the driver informs the engines that they can resume their operation. Engines then restart the faulted transaction. If the new mappings have been correctly installed, the operation

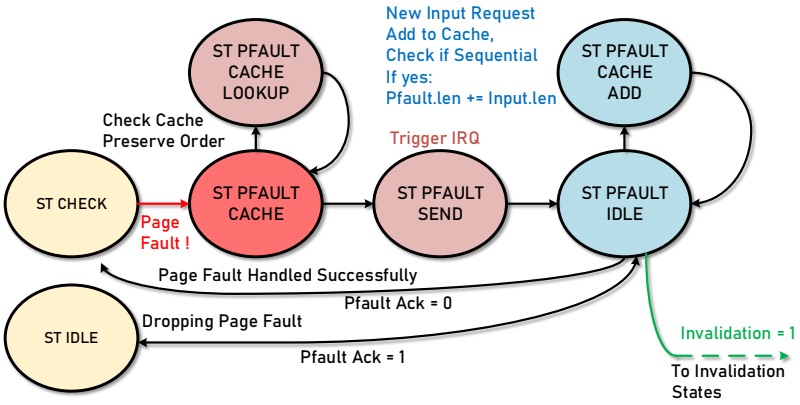

Figure 4.17: State Transitions During a Page Fault.

proceeds in the usual request handling format.

Coyote employs packetization within data requests, which means it frequently receives multiple incoming packets featuring sequentially increasing addresses (parsed packets from initial user requests). Handling each of these packets individually would result in high overhead as it would trigger a new page fault for each packet. To address this, Coyote incorporates an optimization in the form of a special cache buffer that groups these packets together to create larger page fault notifications.

Thus, when a page fault occurs, we will continue reading the next packets that are in the input queue and simultaneously write them into this cache buffer. If any of these packets contain virtual addresses that are sequential to the one that triggered the page fault, we expand the size of the page fault to encompass these additional pages. Consequently, when the driver eventually reads the page fault information, it will encompass a potentially larger amount of pages, spanning across these sequential packets. This coalescing reduces the number of individual page faults that need to be handled.

Following the resolution of the page fault, the subsequent requests are initially retrieved from this request-cache (if there are any, preserving the order of requests), depleting its contents, and then the regular request handling process resumes.

Page faults and invalidations are closely intertwined since both update the state of the TLBs and current mappings. Consequently, careful consideration of locking mechanisms is essential to prevent potential deadlocks or race conditions. In Coyote, a dedicated driver-level lock (`lock_mmu`) is in place. Any operation on the TLBs necessitates obtaining this lock.

However, given the high latencies involved in page faults and invalidations (in the range of microseconds), situations can arise where, for instance, the lock has been acquired by an invalidation initiated from the Linux kernel, while simultaneously, a hardware page fault has occurred. The page fault is issued, but the kernel-level lock is held by the ongoing invalidation. Meanwhile, the hardware's FSM is waiting for the page fault to be resolved, leading to a deadlock. To prevent this scenario, in Coyote, we give priority to the issued invalidations. Therefore, from any stable state, such as `ST_IDLE`, `ST_WAIT_PAGE_FAULT` or `ST_LOCKED`, the FSM transitions to dedicated invalidation states when instructed to handle invalidations. This approach eliminates deadlocks and ensures the efficient operation of the system.

It is also noteworthy to illustrate the amount of overhead incurred when page faults are triggered. In Coyote, page fault handling exhibits a latency ranging from 2 to 3 microseconds on average. If a sufficient number of in-flight transactions can be initiated, this latency can be effectively concealed behind these ongoing operations, thereby requiring only an initial page fault latency to be paid. This aspect becomes important in situations where TLB overflow can arise. For instance, consider a scenario where the amount of mapped memory surpasses the capacity of the TLBs. When such an overflow occurs, a page fault is triggered and handled, while simultaneously, in-flight transfers are in progress. As a result, this page fault becomes concealed behind these ongoing transfers, effectively hiding page fault overheads from the end users.

4.2.5 Unified Memory

Now that we have a better overview of the hardware architecture which is needed to provide the higher level abstractions, we move to software and expore the shared virtual memory model employed within Coyote.

Shared Virtual Memory (SVM) is a memory model that offers multiple devices a shared representation of memory, enabling them to utilize identical virtual addresses. This is possible even when these devices operate independently and lack direct access to each other's physical memory. In the specific scenario of FPGAs integrated into a single host, this implies that both the FPGA and the CPU possess a shared perspective of memory. It's important to emphasize that this shared perspective doesn't inherently require the utilization of identical physical memory. Instead, it denotes an abstract representation created collaboratively by the operating system and the devices engaged in this abstraction.

Shared virtual memory streamlines the exchange of data among multiple execution threads on different devices, eliminating the need for explicit data copying. The primary advantage is the simplification of the programming model, relieving end users from the burden of explicit data management. It also enhances data sharing efficiency by reducing the necessity for numerous data copies in many scenarios. Moreover, the seamless exchange of virtual addresses between devices simplifies the programming of complex data structures (such as pointer rich structures).

As these data management operations are handled in the background, the process of porting applications between platforms becomes considerably more straightforward, as it doesn't require the utilization of device-specific APIs.

Unified Memory is an example of a shared virtual memory in modern heterogeneous architectures featuring discrete GPUs. It facilitates concurrent CPU and GPU access to a unified memory space, optimizing data transfers. This approach allows developers to allocate and utilize memory seamlessly in both CPU and GPU code, simplifying the programming model and enhancing the accessibility and portability of GPU acceleration.

Listing 4.2: Example of processing with FPGA-side memory without unified memory.

```
1  // NO UNIFIED MEMORY
2  void *src_data; void *results;
3  src_data = malloc(size); results = malloc(size);
4  ...
5  fill_data(src_data, ...); computation(src_data, ...);
6  ...
7  void *fpga_data;
8  fpga_data = fpga_alloc(&fpga_data, size);
9  memcpy(fpga_data, src_data, size);
10 host_to_card(fpga_data, size);
11
12 execute_kernel(fpga_data, ...);
13
14 card_to_host(fpga_data, rslt_size);
15 memcpy(results, fpga_data, rslt_size);
16 computation_2(results, ...);
17
18 ...
19 fpga_free(&fpga_data);
20 free(src_data); free(results);
```

Given that FPGA-side memory is often employed in a manner akin to GPU memory usage, the concept in Coyote is to extend this approach to simplify FPGA-side memory management, similar to the way unified memory simplifies memory management for GPUs.

Let's illustrate this simplification with an example: Consider a scenario where some application is deployed on the FPGA to provide acceleration. In this instance, the application works with

substantial datasets and is performing iterative processing. Thus, it could significantly benefit from the lower latency and increased bandwidth of local FPGA-side memory. Consequently, data needs to be transferred to the FPGA-side before processing. The standard code for performing operations like this from the host side can be found in the provided listing. 4.2

To begin, we will initially reserve memory buffers in the host CPU's memory to store our initial data set and eventual results. Some preliminary computations on this data may be performed on the host side at this stage. After completing this initial host-side processing, the data is ready for offloading to the FPGA for acceleration. Unfortunately, due to the absence of data management abstractions in this context, this process must be performed manually.

First, a specific allocation function of some kind must be invoked (`fpga_alloc`), typically triggering a system call to the device driver. The allocation function will reserve a buffer in the device's memory and, at the same time, generate a corresponding memory buffer in the host, ensuring that it is pinned and prepared for transfers to external devices. Subsequently, the data set designated for acceleration must be copied to this pinned buffer using the `memcpy` function. An alternative approach is to pin the already existing data set buffer; in either case, both methods still require a dedicated device-specific API function call at this point.

Listing 4.3: Example of processing with FPGA-side memory with Unified Memory abstraction.

```
1  // WITH UNIFIED MEMORY
2  void *src_data; void *results;
3  src_data = malloc(size); results = malloc(size);
4  ...
5  fill_data(src_data, ...); computation(src_data, ...);
6  ...
7  execute_kernel(src_data, results, ...);
8
9  computation_2(results, ...);
10 ...
11 free(src_data); free(results);
```

With the buffer prepared, we are ready to offload it to the FPGA's memory. This process is again executed through a dedicated API function which will, in turn, issue a system call to the driver (`host_to_card`). Only once this function completes can we start the desired acceleration in the FPGA using the `execute_kernel` function. Once this function is finished, and the FPGA has processed our data, we must reverse these steps. Initially, the data must be retrieved in the reverse direction (`card_to_host`), followed by copying it to the result buffer. From there, we can continue processing the data on the host side. Finally, at the end of the operation, we must deallocate all the device-specific memory resources.

As evident from the previous example, a substantial number of additional device-specific functions are required to facilitate data movement to the accelerator. Now, let's examine another example to observe how the same operation would look like with the unified memory abstraction. The corresponding code for this example is presented in Listing 4.3.

The start is the same as before, we allocate some buffers for our data set and results and perform the initial host-side computation. We then reach a point where we need to invoke our accelerator. As we can see from the code example, in this case, we can execute the kernel on the FPGA-side straight away (`execute_kernel`). No additional management of the memory is needed. Additionally, when the accelerator is done, it can load the results directly into the result buffer (due to the mastering capability on the device). From here onwards, we can continue without any additional device-specific API calls.

From these two small examples, it becomes clear that the code becomes significantly simpler when leveraging the unified memory abstraction. Not only is there a reduction in the number of lines of code, but also, we no longer rely on device-specific APIs for data management. The data also importantly remains consistent even when accessed concurrently by multiple devices. The most significant advantage of this approach, however, is the increased portability of the code between various devices. This simplifies code management and accessibility, particularly in extensive codebases. Furthermore, it effectively conceals device-specific constraints. For instance, even when confronted with data sets that surpass the device's memory capacity, unified memory seamlessly manages overprovisioning, removing the need for explicit user side intervention.

4.2.6 FPGAs Are Not GPUs!

The unified memory model, typically found in the realm of modern GPUs, was successfully adapted for use with FPGAs in Tapasco [116]. This research demonstrated the feasibility of replicating a typical GPU unified memory model on FPGAs. However, it's important to note that the flexibility of FPGAs far exceeds that of GPUs. While FPGAs can indeed be employed similarly to GPUs (catering to specific applications that can leverage this style of compute), this represents only a small subset of the broad spectrum of applications that can take advantage of reconfigurable hardware. Therefore, limiting FPGAs to just a subset of their features by adhering exclusively to the GPU compute model is not the most suitable approach.

In this traditional GPU compute model, data is usually transferred to the GPU's memory and remains there, ideally for repetitive processing on the GPU before being transferred back to the

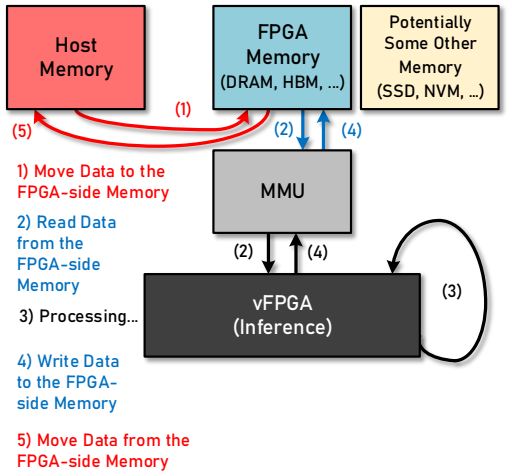

Figure 4.18: Card interface GPU compute model.

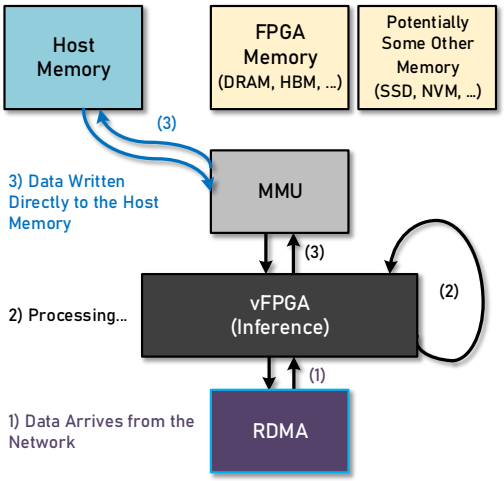

Figure 4.19: Host (Streaming) interface and network interface.

CPU at a later stage. This model is particularly represented in modern machine learning training, where iterative processing over extensive datasets is a frequent requirement.

FPGAs can certainly be employed for similar types of tasks, but they may often lag behind GPUs due to their inherently lower clock speeds. The real strength of FPGAs lies in their adaptability, as they can be customized for various compute models. Consequently, it is imperative to explore how these alternative compute models can be integrated into a shared virtual memory environment.

Consider a scenario where, instead of conducting machine learning training, we need to perform data inference. In this case, the data resides in the host memory initially. If we were to restrict the FPGA to a GPU compute model, we would first be required to transfer this dataset to the FPGA-side memory. Only after the data has been moved to the FPGA-side memory can we begin transferring it to our accelerator (see Figure 4.18). This additional data transfer step directly impacts performance because our ultimate goal is to "stream" data directly from the host memory to the FPGA accelerator without an intermediate stop in the FPGA-side memory.

Afterward, as this data is fed into the accelerator, it undergoes processing, and the results are initially written back to the FPGA-side memory and subsequently transferred to the host memory. Ideally, all of these operations should be fully pipelined, allowing data reading, processing, and result writing to occur simultaneously. This is what allows us to really exploit the pipelined parallelism in FPGAs, an area where they can deliver substantial performance improvements.

As an example of the added flexibility FPGAs can offer, consider including networking in the previous scenario (Figure 4.19). Now, imagine that we need to perform inference in a cloud environment. In this case, data arrives through the network, and the results must be written to the host memory for additional processing. Using FPGA-side DRAM in this context would be highly inefficient. This underscores the importance of having streaming interfaces (along with network interfaces) and the necessity of adapting the overarching memory model to address such scenarios, in addition to the traditional GPU compute model. Offering these additional interfaces maintains the flexibility and vital features of FPGAs while also providing high-level abstractions and ease of use. Coyote, therefore, provides multiple data interfaces to meet the needs of diverse applications deployed on FPGAs:

1. **Card Memory Interface**: These memory accesses are directed toward the FPGA side memory (HBM or DRAM). This memory is primarily employed for high-performance processing, emphasizing the need for maximum throughput and minimal latency, particularly in iterative processing scenarios. The allocation and mapping of this memory among users are managed by the host driver. This interface shares high similarities with the interfaces commonly found in modern GPUs between the memory and processing units.

2. **Host Memory (Streaming) Interface**: This interface allows the end user applications to access the host memory directly. It is special in comparison to a traditonal GPU accelerator. GPUs usually copy the data into off-chip memory prior to processing it. This extra "stop" for the data can lead to excessive overheads, especially if only one pass of the data is necessary. Through this interface, FPGAs in contrast, can directly stream data from and to the host side CPU memory. It is important to note that this interface is only possible through an IOMMU, as the CPU's MMU needs to remain protected.

3. **Network Interface**: Coyote contains an RDMA network stack which provides a networking memory abstraction allowing processes to communicate between remote machines in virtual address space. In this case, it is the MMU's job to make sure the requests contain the virtual addresses, pointing to the buffers, particularly designated for the exchange between two remote nodes. Since this interface is serving remote accesses, it is indirectly integrated into the memory model (and host OS) through the two interfaces mentioned above.

We will now provide a more detailed explanation of these interfaces and describe how they operate. In Section 4.2.7, we will delve into the implementation of these interfaces.

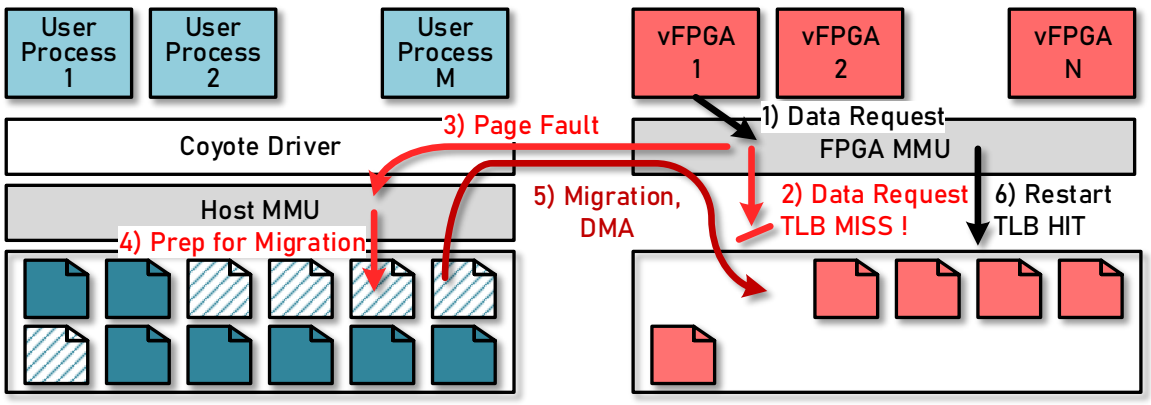

Figure 4.20: Migration of pages to the card memory.

4.2.6.1 Card Memory Interface

The Card Memory Interface offers a closely aligned abstraction to the GPU's unified memory model. It enables user logic in the virtual FPGAs to access data from the local FPGA memory resources, whether it's stored in DRAM or HBM. In the event that the user logic requests data not currently present in the FPGA-side memory, the migration process is initiated without requiring any additional input from the end user. This migration of data is orchestrated through the collaborative efforts of the FPGA-side MMU, host-side MMU, and importantly, the Coyote device driver.

A visual representation of this migration mechanism is depicted in Figure 4.20. The data requests follow a sequence of steps. First, the data requests are submitted and subsequently enter the MMU, where they trigger a TLB miss. It's important to note that at this point, the TLB may potentially contain the mapping for the particular virtual address within the request, but the page fault could still be raised if the mapping is tied to the location in the host memory. Dedicated flags in the TLBs would not match in this case.

The TLB miss event triggers a page fault, which is signaled through the interrupt lines. Upon receiving this interrupt, the device driver reads the page fault status, which includes the virtual address, range and thread information. It then performs a page table walk to retrieve the corresponding pages. As mentioned, these pages may already be mapped, and if so, they are scheduled for migration, and their TLB entries are updated. If they are not mapped, new entries are added to the TLBs, and the pages are migrated in a similar manner. For the migration process, a dedicated migration DMA channel is employed.

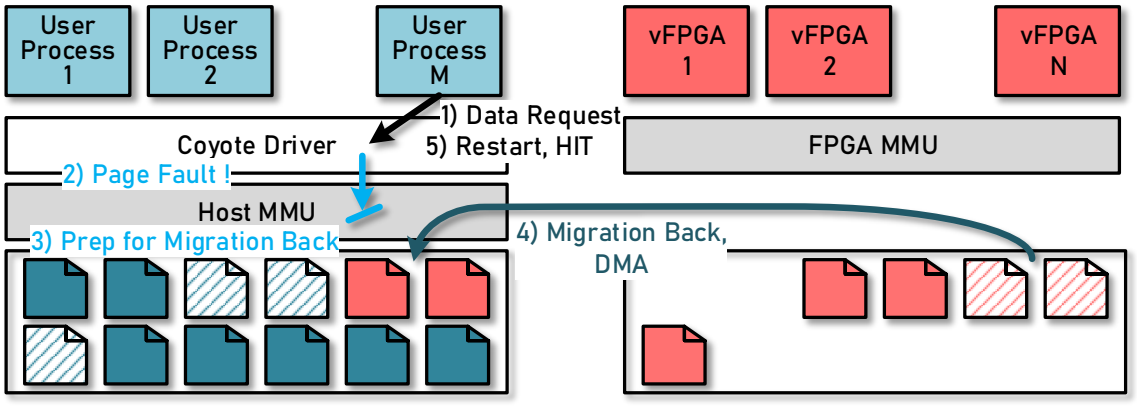

Figure 4.21: Migration of pages back to the host memory.

After migration, the pages that have been migrated are substituted with special device pages. These device pages do not have real memory backing; instead, they serve as indicators that the pages have been migrated elsewhere. In a sense, these pages are now handled as if they were swapped out to a disk.

At this stage, the TLBs are updated and the driver can instruct the MMU to reinitiate the original transaction. This time, a TLB hit will occur, and the operation will proceed as usual. The data will be fetched from the local FPGA memory and provided to the virtual FPGA. From the user's standpoint, apart from the additional latency, the operation remains unchanged, regardless of whether migration occurred in the background or not. The entire operation has been effectively concealed from the end users.

The interesting aspect of this memory model comes into play when an access is made from one of the applications on the CPU side to these migrated pages (see Figure 4.21). In this scenario, a reverse process unfolds. The special device pages, which now stand in for the migrated pages in the kernel, are accessed. This access triggers a CPU page fault, managed by the device driver.

The special device pages contain information about the migrated pages, which is used to facilitate the migration of the pages back, once again via a dedicated migration DMA channel. All the entries in the FPGA TLBs that correspond to these pages will be updated to indicate that these pages now reside in the host memory (same for the host-side MMU). Subsequently, the access from the user application is reinitiated, and the data is provided to the CPU. As in the previous scenario, the entire process occurs entirely transparently to user applications.

This ping-pong operation, characteristic of shared virtual memory, is what plays a crucial role in

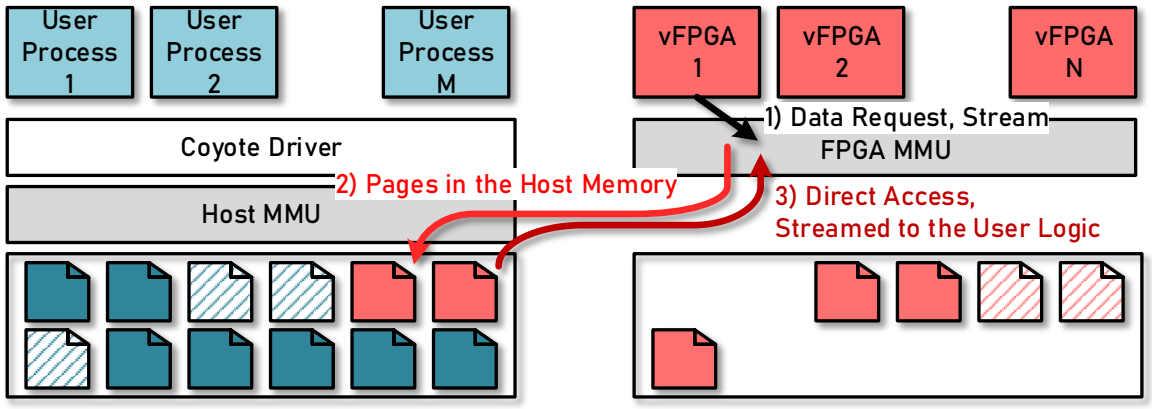

Figure 4.22: Streaming Interface, direct access.

maintaining data consistency in these heterogeneous platforms. Without this mechanism, concurrent accesses from the CPU and the FPGA could potentially lead to data corruption if explicit synchronization is not in place. The absence of the need for this particular synchronization is a significant factor that simplifies interactions with these heterogeneous systems, especially in cases where heterogeneous workloads run alternately on a CPU and an accelerator.

Unified memory's overprovisioning capability is another significant feature. For example, if there isn't enough available FPGA-side memory to fulfill a request, the remainder of the memory is allocated on the host side. This results in memory being split between the FPGA side and the host side, with as much as possible allocated on the FPGA side. Processing begins using the available memory allocated on the FPGA side. At some point, the processing may experience a page fault (when this memory runs out), triggering migrations to bring the new portion of data to the FPGA. The process continues in this iterative manner. The advantage here is that users can make use of the FPGA-side memory even when there isn't sufficient space for the entire dataset.

4.2.6.2 Host Memory (Streaming) Interface

Here, we will provide an overview of the Streaming Interface, which is employed for direct host memory accesses. This interface serves as a key differentiator from the standard unified memory model employed in modern GPUs.

This interface also complements the Card Memory Interface and can be fully utilized concurrently. In fact, these two interfaces can even be employed together, such as for instance conduct-

ing in-memory computations where data is processed as it is read from the FPGA-side memory and written into the CPU-side memory.

Similar to the Card Interface, this interface depends on the FPGA-side MMU for the protection of the host-side memory. The distinction lies in how it operates when a page fault occurs. In such cases, it checks whether the TLB contains the correct mappings and whether the data is available in the host-side memory. If these conditions are not met, a page fault is triggered. However, in this instance, migration may take place, but it will occur in the opposite direction. Any requested page residing in the FPGA-side memory will be moved to the host-side memory.

The manner in which this page fault is handled is similar to the handling of page faults in the Card Memory Interface. The driver performs page table walks to obtain and check all the pages related to the data request. Necessary migrations (or just TLB updates) are then managed. However, a notable difference arises in that the data is no longer read from (or written to) the FPGA-side memory; instead, it is directly streamed to/from the user logic in the virtual FPGA through a dedicated streaming DMA channel of the host CPU-FPGA interconnect.

It is important to note again that no page pinning on the host-side is occurring during all of these processes, if any updates to the TLBs are needed, they will be signaled by the host-side MMU notifiers and handled asynchronously. During these operations consistency of the data will not be violated.

An essential aspect of the Streaming Interface involves page synchronization and potential invalidations. Unlike the Card Interface, where migrated pages cannot be swapped out (as they are already migrated), in the case of the Streaming Interface, as the pages are not pinned, there is a possibility that current mappings within the FPGA may be invalidated by the host system for various reasons (such as swapping out to disk, migrations, etc.).

In such cases, the invalidation process will wait until all outstanding transactions have been completed before the mappings are removed from the TLB on the FPGA side. This approach ensures data consistency as pages in the host MMU cannot be invalidated prior to receiving the acknowledgment of invalidation from the FPGA.

Once this acknowledgment is received and the pages are invalidated on both sides, any subsequent access to these pages from the FPGA will trigger a new page fault, which will then need to be resolved on the host side as any other page fault signaling FPGA-side TLB miss.

4.2.7 Heterogeneous Memory Management

In this section, we will look into the system software responsible for facilitating the shared virtual memory model employed in Coyote. Two critical components within the Linux kernel that enable this functionality are the the MMU notifier interface and the Heterogeneous Memory Management (HMM) API [141].

4.2.7.1 Notifiers and Invalidations

The conventional method for accessing host memory and facilitating data exchange between host and device memory involves explicitly pinning the host pages that need to be accessed by an external device. These pages are typically accessed by the device for additional processing via device-side DMAs. While devices can work with virtual addresses, the actual accesses are performed using physical addresses, with protection provided by the host IOMMU.

The reason for pinning is that the host operating system always retains the capability to swap out pages to disk when memory is running low. Unless the pages are pinned, the operating system does not guarantee that the allocated buffer used by the device will remain in the exact physical location. Once these pages are pinned, they cannot be swapped out or deallocated. This can potentially place a significant burden on the host operating system, as it must manage these pages carefully to avoid disruptions, especially in virtualized environments [39].

An alternative approach is to depend on the kernel for notifying the driver of address space invalidations. Since there is no pinning in this scenario, there is a need to maintain synchronization between the TLBs on the FPGA side and the page tables in the host MMU.

The Linux kernel has introduced MMU notifiers to address the limitations of page pinning, with a focus on virtualized environments. In this setup, the kernel driver registers callback functions for a specific user process. These functions are invoked whenever a portion of the process's page tables is invalidated. In response to these invocations, we can invalidate all entries in the TLBs to keep them synchronized. Importantly, the host MMU cannot complete the invalidations until we return from the callback function.

Given the substantial latency of invalidations, which involves traversing the CPU-FPGA interconnect and ensuring the completion of all in-flight transfers (which can take some time), it's essential that our callback function is capable of sleeping. The callback function is defined within the `mmu_interval_notifier_ops` structure. Significantly, for our application, the call-

back function can sleep, making it a suitable fit for the invalidation framework, as it can be awakened with the interrupt from the FPGA.

Listing 4.4: Invalidation callback function.

```
// Callback function prototype
bool cyt_interval_invalidate(
    struct mmu_interval_notifier *interval_sub,  // Notifier
    const struct mmu_notifier_range *range,  // Virtual address range
    unsigned long cur_seq);
...
// Callback function
if (range->event == MMU_NOTIFY_MIGRATE && range->owner == d)
    return true;

if (mmu_notifier_range_blockable(range))
    mutex_lock(&d->mmu_lock);
else if (!mutex_trylock(&d->mmu_lock))
    return false;

mmu_interval_set_seq(interval_sub, cur_seq);

if (atomic_read(&d->wait_invldt) == FLAG_SET) { // Invalidation completed
    mutex_unlock(&d->mmu_lock);
    return check_invalidation_range(d, range); // Check if retry
} else if (atomic_read(&d->wait_invldt) == FLAG_PENDING) { // Invalidation ongoing
    mutex_unlock(&d->mmu_lock); return false;
} else { // Free to invalidate
    atomic_set(&d->wait_invldt, FLAG_PENDING);
    // Invalidate and go to sleep if possible
    ret_val = tlb_unmap_hmm(d, range, hpid, huge);
    mutex_unlock(&d->mmu_lock);
    return ret_val;
}
```

In Coyote, the primary entities exposed to users are Coyote threads (*ctid*). A single host user process can contain one or more of these threads. As the MMU notifiers are associated with user processes and do not have awareness of Coyote threads, TLBs in Coyote include host process IDs (*hpid*). These process IDs allow us to differentiate between invalidations, as multiple distinct host processes can have the same virtual address ranges and may be concurrently mapped in the TLB of the same virtual FPGA. As soon as a user process is registered in Coyote, it will also be registered to an MMU notifier. To keep track of the registered notifiers and to ensure their proper deregistration when the process exits, this association is maintained in a dedicated hash table.

The code of the callback function triggering invalidations in Coyote is shown in Listing 4.4. A special case where the callback function is invoked is during the HMM migration procedures

within Coyote (`MMU_NOTIFY_MIIGRATE` flag). When pages are being migrated within HMM, the callback will be triggered automatically. However, because these pages are explicitly managed within dedicated driver migration functions, there is no need to further update the TLBs.

Another crucial aspect of the callback is to perform a preemptive check on whether the function can be put to sleep. In a typical operation, the invalidate function can sleep, allowing for a kernel context switch. However, if the function is not blockable (`mmu_notifier_range_blockable`), we issue the invalidation to the device and return from the function straight away. This action indicates that the address range has not been invalidated properly within the device.

In such cases, the kernel must handle invalidations with the awareness that the invalidation of the device pages has not been completed. The kernel can proceed to retry the invalidation of the same address range at a later time, with successive attempts, until the invalidation is successfully executed. Atomic flags within the driver are signalling the current status of the invalidation in hardware (`FLAG_CLR`, `FLAG_PENDING`, `FLAG_SET`).

4.2.7.2 Migrations

The Heterogeneous Memory Management API is a framework within the Linux kernel, primarily aimed at managing heterogeneous memory resources. HMM offers a set of helper functions that revolve around the management of device private pages. These private pages serve as an abstraction of device pages and, as mentioned before, are not directly backed by physical memory. HMM operates with minimal assumptions about the structure of device memory, placing the responsibility for the complete allocation of device-side memory on the device driver.

This aspect aligns well with Coyote's requirements, as it allows for adaptability in structuring device memory to suit our specific needs. For example, within Coyote, certain physical address ranges in device memory may be reserved for retransmission buffers used by network stacks. Additionally, Coyote offers the flexibility to partition FPGA-side memory into different page sizes, aligning with the capabilities of loaded TLBs.

The initial integration of the Heterogeneous Memory Management API into Coyote posed a challenge, primarily due to the diverse page sizes employed within the TLBs. This challenge arose from the fact that huge pages are allocated and pinned in the host memory at system boot time and remain pinned throughout the runtime.

The HMM, on the other hand, relies on the swappability of pages and does not work with pinned pages. This inherent incompatibility with huge pages poses a major obstacle in HMM integration, and it is a critical bottleneck in related work [116]. Without support for huge pages, the

FPGA-side TLBs have limited coverage, resulting in a high number of page faults and consequent performance degradation. Finding a way to overcome this issue became imperative.

Fortunately, in the Linux environment, there is an alternative for managing huge pages backed by a file system. The high overhead associated with pinning these huge pages, the need for additional APIs, and the requirement for explicit user-space management imposed additional burdens on developers and adversely affected application portability. Consequently, a different solution (which is now the prevalent method for allocating huge pages [130]) is the use of Transparent Huge Pages (THP).

Transparent Huge Page (THP) support, as the name suggests, aims to function without demanding significant alterations to the application code. In the THP system, a background Linux thread continually attempts to consolidate individual pages into larger, contiguous blocks, typically 2 MB or 1 GB in size. The entire process operates transparently to the user. This approach eliminates the need for page pinning, allowing users to enjoy the standard advantages associated with huge pages, such as reduced TLB misses.

In the context of Coyote, the use of Transparent Huge Pages is ideal because these pages behave indistinguishably from regular 4 kB pages. In the context of migrations within the kernel driver, these pages can be divided and managed as an array of standard 4 kB pages. Since these pages are still physically contiguous, they can still be mapped as huge pages in Coyote's TLBs. Therefore, the standard HMM API can be utilized with these pages, ensuring the continued coverage of huge pages within the FPGA. The splitting process may introduce some overhead, particularly when migrations involve a large number of individual 4 kB pages. However, since these operations (migrations) are rarely on the critical path, these overheads are acceptable.

We will now briefly outline the migration steps, which make use of the HMM API to handle most of the complex operations in the background. To initiate migration, the process begins with a call to `migrate_vma_setup`. This call accepts a single argument, a `migrate_vma` struct that encapsulates crucial information regarding the pages designated for migration. It provides data about the contiguous virtual address range and the migration direction.. Additionally, it contains arrays that are subsequently populated with the actual page structures that will be migrated. Information for this structure can be supplied either through a page fault or, alternatively, explicit migration can be triggered using the Coyote API from the user space, thereby eliminating the initial page fault overhead, if required. This function prepares the virtual address range for migration by collecting all physical pages and storing them in the `src` array.

Listing 4.5: HMM Migration API.

```
// Structure containing migration information
struct migrate_vma {
    struct vm_area_struct       *vma; // Virtual memory area
    unsigned long               *dst; // Destination pages array
    unsigned long               *src; // Source pages array
    unsigned long               cpages; // Collected pages
    unsigned long               npages; // Number of overall pages
    unsigned long               start; // Starting address
    unsigned long               end; // Ending address
    void                        *pgmap_owner; // Info for migrating back
    unsigned long               flags;
    struct page                 *fault_page;
};
...
// Main migration functions
int migrate_vma_setup (struct migrate_vma *args);
void migrate_vma_pages (struct migrate_vma *migrate);
void migrate_vma_finalize (struct migrate_vma *migrate);
```

It is then possible to check if the pages are pinned. If they are, no migrations will be performed on them. The acquired pages might be fragmented between the host-side and the FPGA-side memories. However, the HMM ensures the data consistency of these pages. This is achieved by examining the pages to determine which pages need to be migrated. Some pages might already be in their correct location, requiring only TLB mapping updates.

Subsequently, additional memory allocation occurs based on the migration direction. When pages are being moved to the FPGA-side memory, the custom Coyote allocator is invoked to provide new pages in the device, if required. In parallel, the allocation of device private pages is carried out.

These device private pages store the physical addresses of the allocated FPGA-side pages in the `zone_private_data` field. They also contain the pagemap struct, which has a field pointing to custom user-defined function within the driver, which will be invoked when the CPU page faults, trying to access these pages. It is within these functions that we trigger the migration back from the FPGA. All of this information (among other) is essential for determining the correct pages for migrations in the opposite direction.

Physical addresses on both sides are then obtained, and the migration is initiated. The migration is carried out by triggering the FPGA's DMA engines to transfer data in the specified direction. Each vFPGA can concurrently initiate a migration, and their DMA descriptors are arbitrated using round-robin arbitration. This approach allows for parallelism among vFPGAs, and, in general, each vFPGA can be managed as an individual device from the driver's standpoint.

The migration process may take some time, and interrupts are employed to signal the completion of DMA operations. Once the interrupt is received, TLBs are updated with new mappings (old ones get invalidated). The `migrate_vma_pages` function will then perform the actual swap of the source pages to the destination pages. Finally, the host-side MMU will be updated with the new page information and unlock the pages, effectively ending the procedure (`migrate_vma_finalize`).

4.2.8 Evaluation of the Shared Virtual Memory

Next, let's assess the Shared Virtual Memory model utilized in Coyote. In the modern landscape, machine learning stands out as a prominent domain characterized by two fundamental processing phases: training, in which models acquire knowledge from labeled data, and inference, where these trained models provide predictions for new data. These two computations are exceptionally well-suited to showcase Coyote's adaptability, particularly when compared to other state-of-the-art systems that adhere to strict computation models.

4.2.8.1 Machine Learning Clustering

Let's begin with K-Means, a well-known illustration of unsupervised machine learning. Like most machine learning training algorithms, this is a computationally demanding procedure that necessitates optimizations across multiple iterations. Therefore, it benefits greatly from high data throughput and minimal latency to ensure efficient performance. In general, modern GPUs are well-suited for such training tasks due to their parallel processing capabilities and high memory bandwidth. Additionally, FPGAs can be configured to effectively support machine learning training, despite operating at lower clock speeds. They offset this limitation by employing deep processing pipelines, resulting in improved overall performance [211, 120].

It's worth emphasizing that our intention in this example is not to advocate for the use of FPGAs in machine learning training. That task is better suited for other research and discussions. In this context, our aim is to assess whether the unified memory abstraction is a suitable approach for these types of workloads in a general sense.

It's important to remember that the primary advantage of this abstraction is its ability to simplify the end-user interface. Our objective here is to investigate whether the cost paid for this abstraction (demand paging) is acceptable in the context of workloads that fit these patterns.

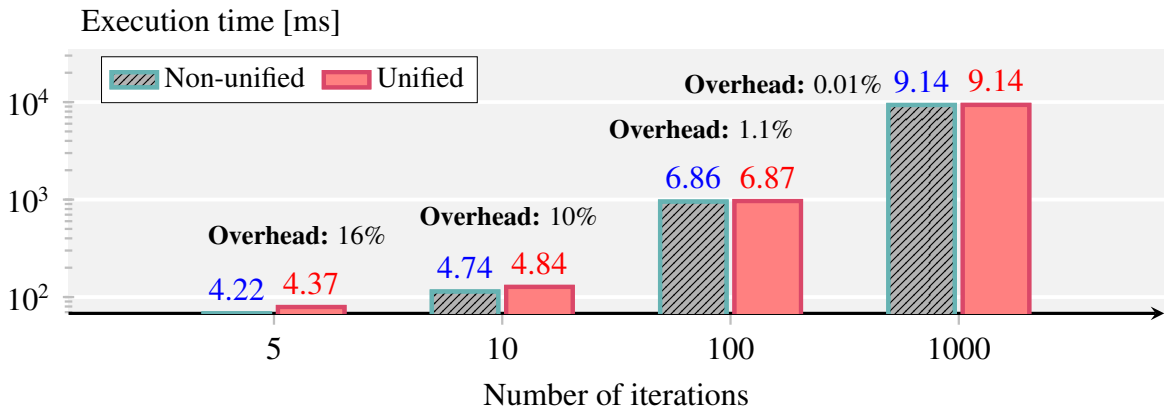

Figure 4.23: Unified memory overhead for K-Means clustering.

We have chosen to integrate a K-Means clustering algorithm as an example of a widely-used machine learning training method. K-Means is clustering technique that groups data points based on their similarity. The process begins by selecting K initial centroids, followed by assigning each data point to the nearest centroid. After that, it recalculates the centroids, repeating these steps for a fixed number of iterations or until convergence is achieved. The outcome is the formation of K clusters that minimize the squared distances between data points and their respective cluster centroids. Due to its iterative nature and substantial computational demands it imposes, this algorithm is ideally suited for execution with FPGA-side memory.

For the K-Means hardware operator, we choose an open-source bit serial K-Means operator implemented specifically for an Intel HARP platform [106, 103]. The porting is fairly straight forward and a lot of boilerplate code present in the original repository particularly designed to support the operation on HARP is removed. The experiment is executed on the Alveo-u55c board [15] with 16 GB of HBM attached.

At the beginning of the operation, we allocate and prepare three distinct buffers: one for the initial centroids, one for the dataset, and another for the results. The CPU initially populates these buffers with data, and they are then transferred to the FPGA-side memory for further processing. The unified memory abstraction seamlessly manages the allocation of these buffers and the data transfer to the FPGA-side memory, simplifying the process. In contrast, when using the baseline approach without unified memory, the management of memory and data movement needs to be explicitly handled, as detailed in Section 4.2.5. After the data is transferred to the FPGA memory, it is streamed to the hardware operator where the primary processing takes place. This processing is carried out iteratively for a predetermined number of iterations. Once the entire operation is completed, the resulting clusters are returned to the host side.

The objective is to assess the overhead associated with this entire operation when relying on the unified memory abstraction for data management, as opposed to the conventional approach of manually managing the data. The results of the K-Means algorithm, for these two scenarios, across different number of iterations, for a dataset size of 128 MB, are shown in the Figure 4.23.

As evident from the results, even with a small number of iterations and a relatively modest dataset, the overhead incurred by using Unified Memory and demand paging is around 10%. This cost is primarily attributed to page faults on both sides, the need to remap pages by modifying page tables, and the data copying between the host and FPGA. In practice, this cost is distributed over the number of iterations the computation performs for each data transfer. As a result, for a higher number of iterations, the overhead becomes nearly negligible.

This is particularly significant since these interfaces are designed to cater to large machine learning models with substantial datasets that require a significant number of iterations to complete (far greater than in this example). Consequently, the abstraction of Unified Memory significantly simplifies development and reduces the need for user involvement in data management, all while imposing minimal overhead on targeted applications. The resource usage of the K-Means operator is shown in the Table 4.4.

Table 4.4: K-Means resource usage (Alveo-u55c).

Configuration	CLBs	LUTs	Registers	Block RAM
User shell	53 467 (32.81 %)	253 399 (19.44 %)	337 595 (12.95 %)	1034 (51.29 %)
K-Means (1-engine)	22 830 (14.01 %)	114 859 (8.81 %)	123 548 (4.74 %)	900 (44.64 %)

4.2.8.2 Machine Learning Inference

The model of computation that relies on FPGA-side memory is common in modern FPGA shells (while lacking the shared virtual memory). However, it's important to note that certain algorithms do not benefit from this model and may require a different approach to achieve optimal performance.

Let's stay within the realm of machine learning to illustrate how even within a single field, different computation requirements can arise. Instead of focusing on training, we'll shift our attention to inference. Inference, in the context of machine learning and artificial intelligence,

involves using a pre-trained model to make predictions or draw conclusions from new data. Inference is a critical step in deploying machine learning models for real-world applications and typically has lower latencies compared to the training phase. FPGAs excel in inference tasks, thanks to their ability to handle processing within deep pipelines efficiently [80, 206, 154].

In this scenario, we opt for the decision tree algorithm. Decision trees are a commonly used form of supervised machine learning, suitable for tasks such as classification and regression. They are built through a recursive process of dividing the data into multiple groups. A cost function is employed to evaluate potential splits, and a greedy algorithm selects the best split candidate at each step. Recursive splitting continues until a predefined tree depth is reached to prevent overfitting.

The integrated application is an open-source [165] implementation of Gradient Boosting Decision Trees [153], with a specific focus on performing inference over decision tree ensembles. This application is well-established and has been deployed in real-world scenarios, particularly for scoring flight routes [166]. To assess its performance, we are conducting a comparison of the inference execution in Coyote on different platforms with the results obtained on Amazon F1 instances, utilizing the Xilinx SDAccel programming framework [7], and the Intel HARP environment [162].

In our performance comparison, we assess the inference throughput (measured in scored tuples per second) on Coyote in comparison to the same application running on the Amazon F1 platform and on Intel's hybrid CPU-FPGA HARP v2 platform. We perform the test on both the vcu118 development board and the Enzian platform. Notably, both of these platforms incorporate the same chip as the Amazon F1, making them an appropriate basis for comparison. It's important to highlight that the design on the Enzian platform attained a higher clock frequency (300 MHz) compared to the designs on the vcu118 and the F1 instance (250 MHz), likely due to the use of a chip with a superior speed grade. The HARP v2 was more limited in this respect, with the FPGA design clocked at 200 MHz.

Given that the F1 platform is designed for OpenCL applications, the SDAccel port adheres to a strict GPU-based compute model, resulting in higher data transfer overhead. In all scenarios, we evaluate the throughput with both one and two instances of the application running on the FPGA, each processing a dataset with 4,000 tuples. The inference engine across all platforms is compute-bound and requires just 4 GB/s of memory bandwidth, enabling two instances to operate at full capacity.

The results are shown in Figure 4.24, while the resourse usage is shown in 4.5. Significantly,

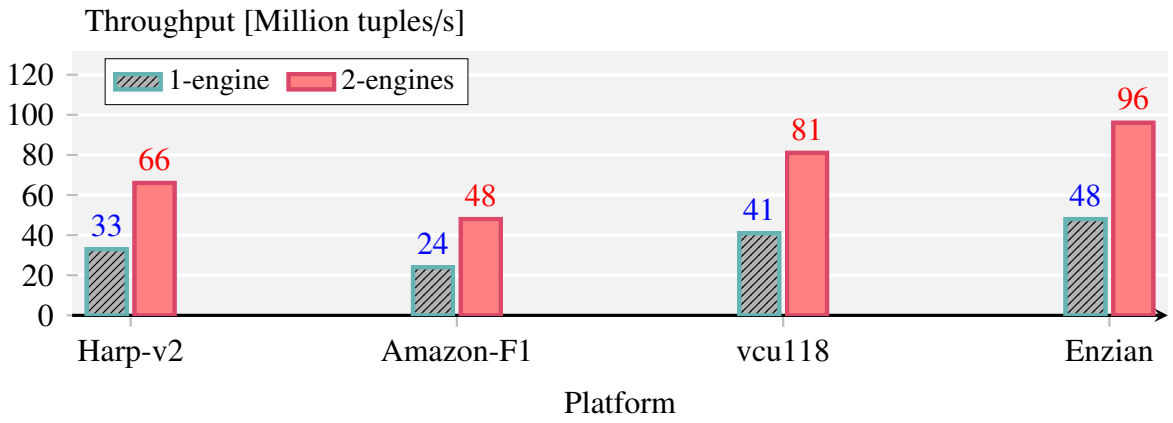

Figure 4.24: Gradient boosting decision trees.

Coyote exhibits comparable or superior performance when compared to the two commercial baselines. It's essential to note that the primary focus of this comparison is the F1 instance, and the results show that the F1 platform delivers the lowest performance among all the platforms. The main contributing factor to this performance gap is the strict compute model, as discussed earlier. This highlights how even a robust cloud platform can experience reduced performance for a significant subset of applications due to the limitations of the compute model.

In addition, it's important to highlight that Coyote offers portability and supports multiple tenants, allowing for a more versatile usage. In contrast, SDAccel limits the usage to a single tenant and single dynamic region on the FPGA. On the other hand, HARP provides direct access to host memory, similar to Coyote, but it is restricted to only this specific type of access without the capability to efficiently utilize FPGA-side memories. Similar to the F1 platform, HARP lacks other abstractions like multi-tenancy.

These examples highlight the critical importance of flexibility in the compute model within the shells of these devices. Such flexibility is necessary to address a wide range of modern compute challenges and accommodate different types of algorithms effectively.

Table 4.5: Gradient Boosting Decisiont Trees resource usage (Alveo-u55c).

Configuration	CLBs	LUTs	Registers	Block RAM
User shell	38 671 (23.73 %)	188 574 (14.46 %)	236 029 (9.05 %)	616 (30.56 %)
Decision Trees (1-engine)	21 103 (12.95 %)	109 251 (8.48 %)	118 596 (4.55 %)	505 (25.05 %)

4.3 Network Services: TCP/IP

FPGAs are commonly viewed as straightforward accelerator devices linked to the host processing system through conventional interfaces like PCIe. However, modern computing is heavily intertwined with network interactions, where data arrives from numerous locations for processing. Therefore, the setup in which the FPGA's sole connection to the external world is through the PCIe link is suboptimal.

The primary challenge lies in the excessive data movement, which is a significant source of overhead in modern computing [152]. Whenever data intended for FPGA acceleration is received from or transmitted over the network, it necessitates additional detours through the host CPU memory. This, in turn, demands additional CPU involvement to coordinate the entire interaction with the accelerator and manage the data transfer to and from the FPGA. This often results in inefficient utilization of the CPU's processing capacity, which could otherwise be employed for more productive tasks. The cumulative latency overhead resulting from this extra data movement can be substantial [184, 126].

The solution to this issue is to eliminate the dependency of FPGAs on the host CPU. Instead of directing data from the network straight to the CPU at the outset, the data can be initially routed through the FPGA. This approach is made feasible by the abundant I/O and high-performance transceivers available on modern FPGAs, which can directly connect to high-speed network traffic. Consequently, the FPGA can function as a "bump-in-the-wire" for network data, allowing it to carry out its acceleration tasks with minimal additional data movement.

However, to effectively work with data in modern data centers and cloud environments, it's essential to operate at a much higher level of abstraction compared to the lower physical layers of the networking stack. Otherwise, the FPGA would lack awareness of the actual processing and data structures used in these network communications. To address this, dedicated high-performance network stacks must be implemented in the FPGA fabric. These stacks have to be designed to handle the line rates encountered in modern distributed computing.

Fortunately, there is existing work in this domain, and capable high-performance network stacks designed for modern data-center FPGAs have been developed. One example of such a stack is the open-source FPGA network stack that facilitates high-speed TCP/IP communication [186]. This stack, entirely developed using high level synthesis, delivers complete and dependable TCP/IP communication directly to the FPGA fabric.

The TCP/IP protocol is important primarily because it forms the foundation of internet and

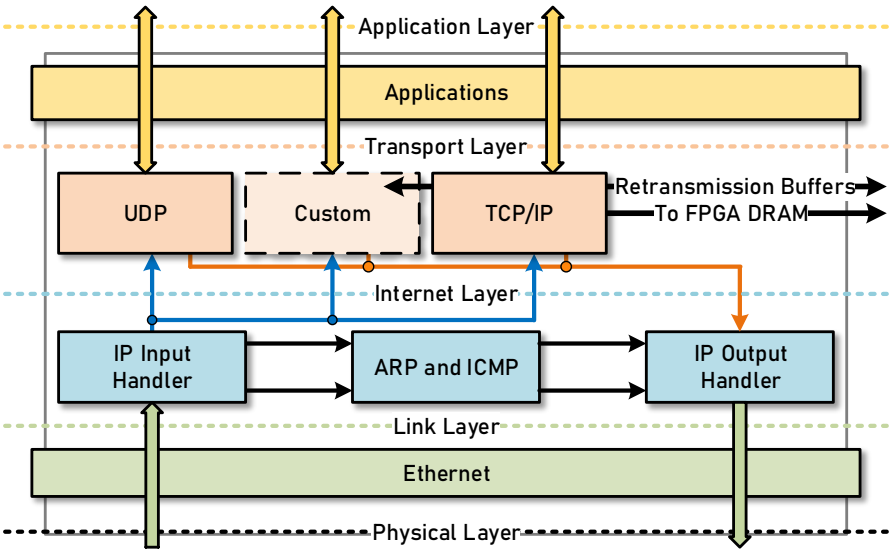

Figure 4.25: Network Stack Architecture, as shown in [185].

network communication today. By directly embedding support for this protocol into FPGAs, these devices can seamlessly integrate into modern networking architectures. This streamlines the process of harnessing their acceleration capabilities, as it eliminates the need for extensive efforts to provide compatibility. In the context of Coyote, the TCP/IP stack is offered as a service to each of the tenants. This provision allows Coyote to grant each tenant access to a high-performance TCP/IP interface, enabling every tenant to conduct real-time computation on network data as it seamlessly traverses the system.

4.3.1 TCP/IP Architecture and Interfaces

The architecture of the open-source TCP/IP stack is extensively described in [185]. It comprises several modules responsible for managing different layers of the networking stack (from link layer all the way to transport layer). Notably, since all of these modules are pipelined in both send and receive directions independently, they operate in parallel, enabling the stack to fully leverage the performance capabilities of the FPGA, reaching speeds of up to 100 Gbps.

When this stack is deployed within FPGA fabric, it maintains complete modularity, enabling it to be customized for varying feature sets and functionalities. This is particularly evident in its capacity to scale the number of connections by dedicating more on-board resources.

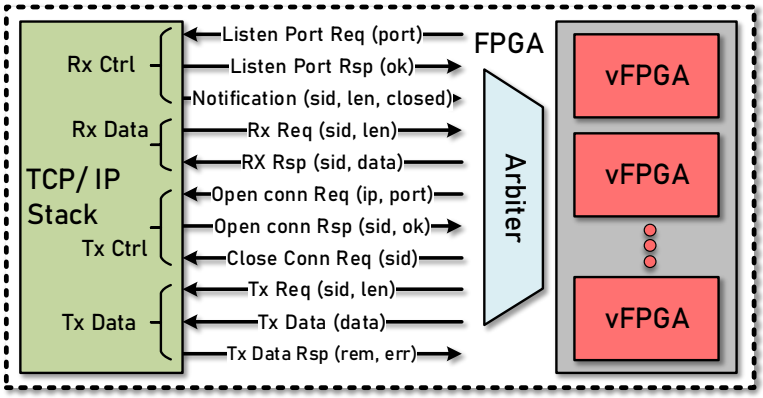

Figure 4.26: TCP/IP Interfaces.

The stack also ensures dependable communication, which is vital in data-center clusters where network congestion is a frequent occurrence. To support this reliable communication, it's essential to have a substantial external memory capacity to store buffers in case of data retransmissions. Since the on-chip memory of an FPGA is often insufficient for this purpose, the network stack must be equipped with a dedicated pathway to store retransmission data in the FPGA's off-chip memory (either DDR or HBM).

The stack distinctly separates various networking layers, a fundamental aspect that facilitates the development of additional networking protocols by leveraging existing ones. This separation enhances modularity and serves as the foundation for the exploration of additional protocols on FPGAs. One such protocol of significance for Coyote in this context is RDMA.

The interfaces the TCP/IP stack exposes are shown in 4.26. The structure of these interfaces is a priority in Coyote, as these interfaces need to be modified to function effectively in a multi-tenant environment, ensuring that they offer fair quality of service to all tenants. Here are brief descriptions of these interfaces:

- Listen Port request and response interfaces are located on the Rx (Receiving) side of a network. Its primary function is to activate a specific port, enabling it to enter a listening state. Once the port is in this state, external entities can establish TCP connections to it, enabling data transfer.
- The Notification interface serves as a mechanism to alert the user logic about incoming network packets. These notifications relate specifically to packets that have been placed into receive buffers designated for a particular connection session ID.

- Rx requests and responses are interfaces dedicated to receiving data. In this context, requests are typically initiated through this interface by user logic after receiving a notification that data is present in receive buffers.
- Connection management interfaces are used for the purpose of establishing and controlling connections with remote TCP/IP nodes. If the connection establishment is successful, the response will include a session ID, which is subsequently employed to identify, manage and eventually close the specific connection.
- Tx requests and responses are designed for sending data. They serve as counterparts to the Rx (Receiving) interfaces. Through these Tx interfaces, users can transmit data to external connections, each represented by its respective session ID.

4.3.2 Integration of the TCP/IP Stack

Within the Coyote system, there's a requirement for an additional arbitration layer to manage multi-tenancy effectively. It's crucial to ensure that incoming and outgoing network packets are correctly directed to and from the respective tenant for each data request. Furthermore, each of these packets must maintain the thread abstraction within Coyote.

This implies that it's not only necessary to route the data to the correct vFPGA, but also each data word must include an additional *ctid* field representing the appropriate Coyote thread ID for both the inbound and outbound data transfers. Simultaneously, we must uphold the quality of service, and it's imperative that each tenant can achieve a fair share of the overall TCP bandwidth. The architecture of the TCP arbiter is shown in Figure 4.27.

Since the port-activating interface is inherently in-order, there is no need to propagate Coyote-specific IDs (*vfid*, *ctid*) all the way to the network stack. A straightforward round-robin arbitration scheme for each tenant is sufficient to guarantee consistent QoS. The IDs associated with the user granted permission by the arbiter, are recorded in the queues. Subsequently, when responses are received, we can extract this user-specific information, enabling us to accurately direct the response to the appropriate destination.

We will also retain a mapping at this point between the port and the Coyote-specific IDs. This information will prove valuable in the future, especially when an external connection is initiated to a specific port. It helps in determining the destination vFPGA that should be notified when this connection decides to send data.

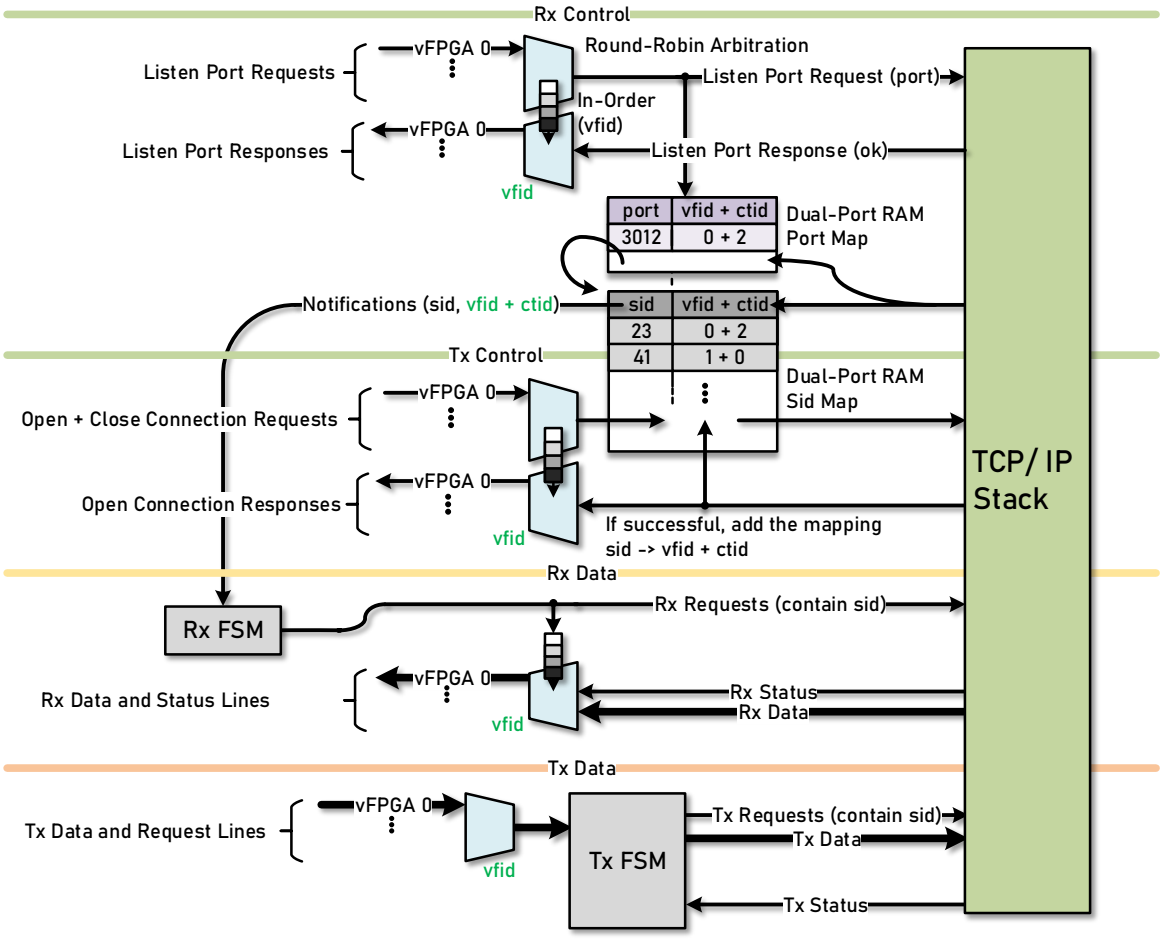

Figure 4.27: TCP/IP interleaving and arbitration.

The same principle extends to the connection management interface. The initiation of connections is subject to arbitration among all vFPGAs, and the corresponding responses are matched with the information stored in the dedicated queue to ensure proper handling. We streamline the process additionally by combining separate interfaces for opening and closing of connections into a single one, treating them as a union. This approach reduces the overall number of bits that the complete TCP interface needs to expose.

In the case of the notification interface, the process becomes somewhat more intricate. Here, notifications arrive with session IDs originating from remote nodes that lack awareness of Coyote-specific identifiers. Consequently, it's essential to establish a mapping between these session IDs and the corresponding Coyote IDs in order to designate the appropriate destination.

This mapping can be initially created when a connection is being opened through a dedicated opening request. This interface carries Coyote IDs that can be associated with the session ID in

the response, provided the connection opening is successful. Additionally, the mapping can be added if a remote node initiates a connection to a specific local port in the opposite direction. In such a scenario, the port information is conveyed through the extension within the notification interface. The lookup is then executed in the port table to identify the correct destinations, which are subsequently incorporated into the session ID table to complete the mapping.

Finally, a simplification of the Tx (transmit) and Rx (receive) data interfaces has been introduced. Rather than relaying notifications about received data to the end users, we now respond promptly by initiating a data request as soon as notifications appear. These data requests and responses are executed in an orderly manner, ensuring they are routed to the appropriate vFPGA as indicated in the notification. Any potential backpressure stemming from user applications in this scenario is managed at higher levels of the stack, primarily through a credit system.

4.3.3 Resource Usage of the TCP/IP Stack

The TCP/IP brings a lot of potential use cases as it allows us to create connections by using the most common protocol in the modern world. However, as with everything, this doesn't come for free. The resource usage of the TCP/IP stack in the shell is shown in Table 4.6.

The utilization of CLBs by TCP amounts to approximately 20% of the total resources within the chip. This represents a significant allocation of FPGA fabric dedicated to the network stack. Additionally, it necessitates the establishment of a dedicated memory channel to handle retransmission buffering on the FPGA side. These memory channels are typically positioned in the central area of the devices [9, 8, 10], or at the bottom in the case of HBM [10, 14, 15]. In contrast, the network transceivers are located at the periphery of the FPGA. This architectural arrangement can result in routing extending over a broad area, which could be a major source of congestion.

Table 4.6: TCP/IP stack resource usage (Alveo-u55c).

Module	CLBs	LUTs	Registers	Block RAM
TCP stack	32 657 (20.04 %)	185 513 (14.23 %)	205 981 (7.9 %)	152 (7.54 %)

Nevertheless, the trade-off here is acceptable due to the immense potential of applications and interactions that these devices can harness when granted direct access to external networks.

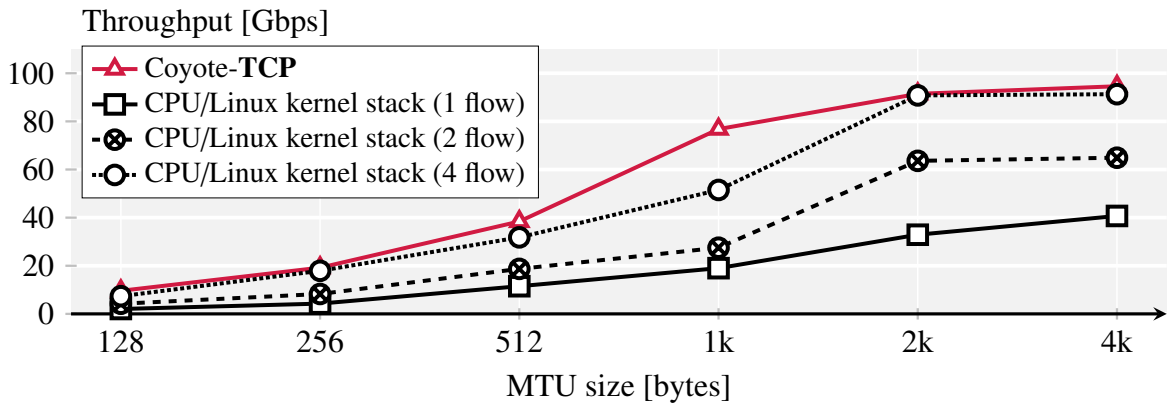

Figure 4.28: TCP/IP throughput comparison.

4.3.4 Evaluation of TCP/IP

In this section, we will evaluate the performance of the TCP/IP stack. Our analysis will involve a comparison between the fully offloaded TCP/IP stack in Coyote and the traditional CPU-based Linux kernel TCP/IP stack.

The experiments take place in a cluster comprising in total of 10 nodes, each equipped with four Intel Xeon Gold 6234 processors and 376 GB of memory. The FPGAs employed for these tests are Alveo u55c FPGAs, which are connected to each host CPU via PCIe with 16 lanes. These CPUs and FPGAs are interconnected via a high-speed 100 Gbps Cisco Nexus 9336C-FX2 network switch. This configuration provides a robust and high-performance environment for conducting the experiments.

First, we examine the results of the throughput test, as illustrated in Figure 4.28. As depicted in the figure, a single standard CPU-based TCP/IP flow is not as efficient as a single FPGA flow, and the performance difference is quite significant. The CPU is only able to achieve a throughput of approximately 40 Gbps at larger MTU (Maximum Transmission Unit) sizes. Consequently, a substantial portion of network bandwidth remains unused in this scenario. To saturate the network using a CPU, we had to run four concurrent flows. This exerts significant pressure on CPU cores, essentially rendering it ineffective for any other concurrent tasks apart from networking. In a multitasking environment, this implies that a substantial portion of CPU cycles will be wasted, resulting in a notable degradation in the overall performance.

At the same time, the latency figures, as depicted in Figure 4.29, reveal even more pronounced differences. Particularly, for smaller MTU sizes, the disparities in latency are evident. The added data movement overhead within the CPU results in significantly higher latency numbers.

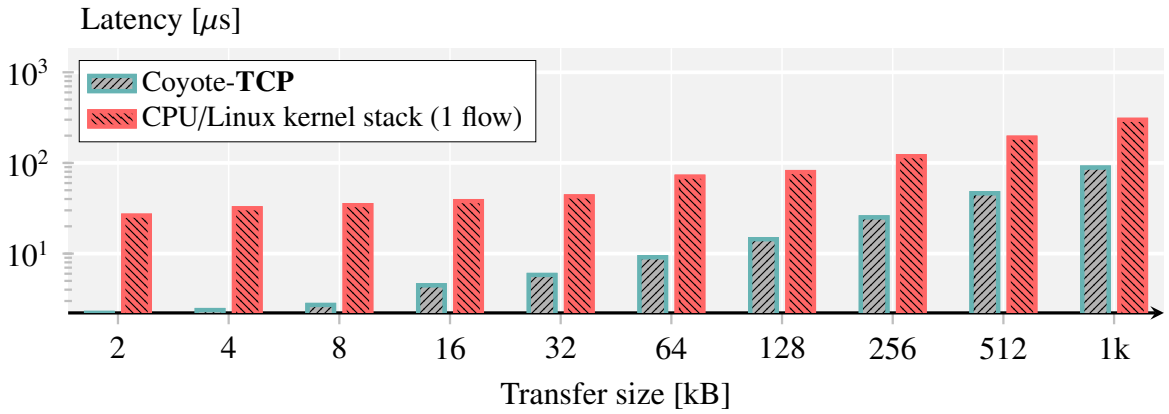

Figure 4.29: TCP/IP latency comparison.

While there are software-based optimizations available that can expedite TCP/IP processing, such as DPDK [11], it's important to note that the processing remains CPU-bound. This means that the CPU is still heavily involved, diverting valuable computing cycles from other tasks. This is where the offloaded TCP/IP network stack demonstrates one of its primary advantages—it manages the entire processing without the need for additional involvement from other devices.

Consequently, the offloaded TCP/IP network stack can directly deliver data at the transport layer to the FPGA fabric, enabling FPGA-based applications to process it directly. This forms the first foundation for the implementation of "bump-in-the-wire" type of processing, where the FPGA seamlessly intercepts and processes the data without the need for CPU intervention.

4.4 Network Services: RDMA

Now, let's delve into another networking service provided by Coyote: Remote Direct Memory Access (RDMA). RDMA embodies a networking access pattern that offers a higher level of abstraction in contrast to the TCP/IP. It fundamentally serves as a memory abstraction operating at virtual address layer. This makes it a considerably more appropriate choice for Coyote when compared to the TCP/IP stack.

RDMA greatly improves data transfer capabilities between remote machines. In contrast to conventional approaches where the CPU is responsible for data transfer, RDMA enables two remote machines to communicate by giving them the ability to directly read and write data to each other's memory. The CPU is relieved of the responsibility for handling data transfers since all data movement is delegated to a dedicated Network Interface Card (NIC).

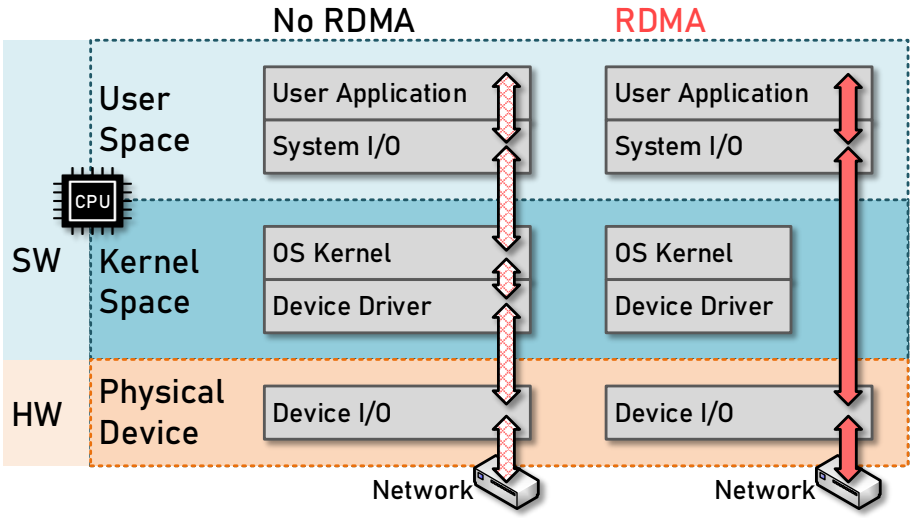

Figure 4.30: RDMA stack.

One of the most compelling features of RDMA is its ability to enable two user processes to communicate at a very high level of abstraction with minimum overheads. They can achieve this by directly sharing data using their user space pointers. Importantly, throughout these operations, RDMA completely circumvents the operating system kernel, effectively eliminating most of the overheads that can stem from it. Additionally, all transfers are executed using zero-copy mechanisms, eradicating any redundant data movement. This results in efficient and rapid data transmission between processes. Essentially, RDMA offers a streamlined and highly efficient method of data communication, rendering it an optimal choice for situations where swift, low-latency data transfer is of critical importance. RDMA finds extensive use in high-performance computing clusters, data centers, and cloud environments [164, 79, 133].

In addition to the advantages of reduced latency, RDMA also offers high throughput, accommodating high-speed data transfer rates. This capability holds significant importance in modern data centers and clouds, where network communication speeds are continually on the rise, and 400 Gbps networks are already becoming available. In order to achieve optimal performance at such high speeds, it is crucial to have a dedicated network processing module. This relieves the CPU from the need to handle this processing (which may well not be possible at these speeds).

How can FPGAs leverage this networking abstraction? One approach is by establishing a direct connection between the FPGA and an RDMA-capable NIC. In this configuration, the FPGA is situated between the NIC and the host memory, essentially transforming the entire system into a

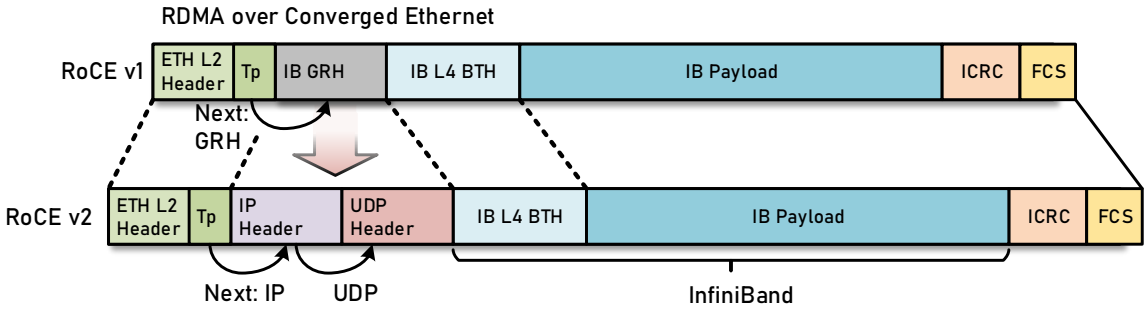

Figure 4.31: RoCE protocol.

highly adaptable smart NIC. This arrangement allows for enhanced data processing capabilities and efficient data transfer between the network and the host memory. Unlike the TCP/IP stack, which lacks memory abstractions and requires additional management for network packets to interface with memory and other system components, in this context, network packets traveling to or from the FPGA can be directly processed as they are written to or read from the main memory. This approach achieves a true ”bump-in-the-wire” processing, where data is processed on-the-fly without the need for any additional data movements or delays.

However, off-the-shelf smart NICs (combining FPGA and RDMA-capable NIC [150]), come with fixed network processing capabilities due to fixed design of the networking card, offering limited modularity. This imposes constraints, especially when attempting to explore additional abstractions within the system (as is the case in Coyote). For example, exploring multi-tenant RDMA environment would be very difficult with fixed NIC interfaces in this configuration.

An alternative is to integrate the RDMA NIC directly into the FPGA fabric. FPGAs come equipped with high-speed transceivers and are well-suited to meet contemporary networking speeds and demands (as demonstrated with the TCP/IP stack). This approach offers the advantage of complete flexibility for the stack, enabling the adaptation of interfaces as required and even providing room for further exploration in network packet processing, such as the addition of extra one-sided or two-sided verbs.

The network stack already integrated into Coyote at this point (the TCP/IP stack) is highly modular, allowing for easy addition and expansion of networking layers. One such implementation [185] brings RDMA functionality with RoCEv2 (RDMA over Converged Ethernet, Figure 4.31). In RoCEv2, network messages are encapsulated within standard UDP packets, resulting in a simplified transport layer compared to previous version. RoCE incorporates the

InfiniBand communication standard encapsulated within UDP packets. It supports both traditional two-sided verbs (SEND), involving remote CPUs, and RDMA-specific one-sided verbs (READ, WRITE). The latter resemble traditional load and store instructions over the network, offering high performance and low latency. The implemented RoCE stack operates at 100 Gbps, keeping pace with modern FPGA networking speeds.

RoCE combines Ethernet for network connectivity with RDMA for direct memory access, in contrast to earlier RDMA implementations that relied on a higher-overhead TCP/IP transport layer (iWARP [75]). RoCE simplifies the transport protocol, replacing the TCP stack with a more straightforward transport protocol like UDP. This simplification results in higher performance, lower latency, and greater scalability. RoCE also performs exceptionally well in virtualized environments [151], making it an ideal fit for Coyote.

4.4.1 RDMA Architecture and Interfaces

The architecture of the integrated RDMA stack, depicted in Figure 4.32, shares a significant portion of the lower networking layers, with the existing TCP/IP stack. This commonality extends all the way up to and including the IP layer. Above this layer, the already established UDP is used as the foundation (transport layer) for the RDMA stack. Notably, this shared lower layer structure allows both the RDMA stack and the TCP stack to coexist on the same device, utilizing the same networking port. While this is technically feasible in Coyote and bitstreams can be built with this functionality baked in, it's important to acknowledge that, in practice, the complexities of routing and increased resource consumption may limit its utility. This may become a more viable option in the future as FPGAs continue to grow.

The integrated base RDMA stack in Coyote features parallel receive and transmit paths, ensuring optimal performance in both data transfer directions. Network packet headers are parsed in sequential pipelining stages, utlizing the benefits of pipelined parallelism. In this regard, RDMA stack is similar to the already covered TCP/IP stack.

The initial InifiniBand header parsed is the Base Transport Header (BTH), which contains essential queue pair information, encompassing all details related to a specific connection. Notably, in the context of Coyote, the queue pair number (QPN) is used to encode additional critical information during creation, enabling the mapping of queue pairs to their respective tenants and thus establishing a true multi-tenant RDMA environment. The BTH also holds packet sequence numbers (PSN), vital for correctly ordering packets from various queue pairs and ensuring reliable communication, a topic we'll explore further later on.

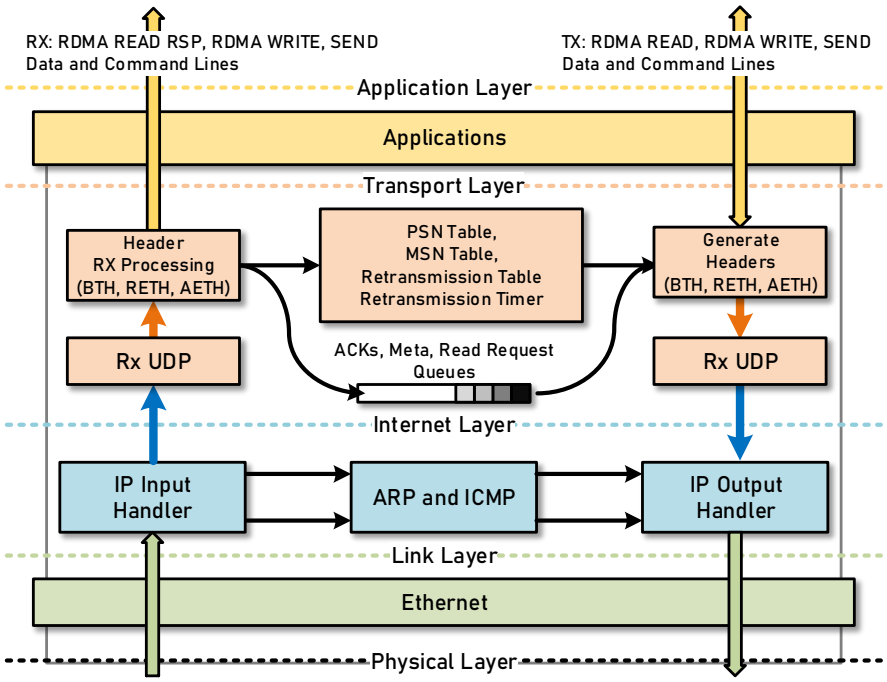

Figure 4.32: RoCE stack.

Following the BTH, the RDMA extended headers (RETH) provide details about the actual data exchange. These headers include crucial user space virtual addresses specifying where data will be read from or written to. Importantly, certain fields within this header are utilized to encode Coyote-specific information as well.

Finally, the AETH (ACK Extended Transport Header) header contains all the necessary information to acknowledge network messages between remote machines. A comprehensive explanation of the RDMA pipeline architecture can be found in [185].

The base stack initially incorporates only one-sided RDMA verbs, specifically RDMA READ and RDMA WRITE. As their names imply, these verbs are tailored for typical memory access at the endpoints and, in such operations, excel by delivering exceptional performance with low latency and high throughput.

However, our aim is to facilitate a more extensive array of operations, particularly as we incorporate accelerators into the system. To achieve this, we additionally introduce support for the traditional two-sided SEND verb. Despite its two-sided nature, this verb presents intriguing applications within our specific system, which we will explore in one of the next sections 4.4.5.

It's crucial to underscore that our choice to adhere to the SEND verb is intentional, driven by

our commitment not to introduce verbs that are not inherently supported by typical commercial RDMA NICs. Ultimately, our objective is to maintain complete compatibility, facilitating efortless communication between these RDMA NICs and Coyote. This enables inter-cluster communication and the seamless integration of FPGAs into the broader data center hierarchy.

We will now look into the interfaces the RDMA stack exposes. These are shown in Figure 4.33. Similarly to the TCP/IP stack, these need to be modified to function effectively in a multi-tenant environment. Here is the brief description of these interfaces:

- The Send Queue interface serves as the primary method for issuing commands to the RDMA stack. Through this interface, users can initiate any of the available RDMA stack operations, such as reads, writes, and sends. This interface describes the buffers involved in data exchange, using standard virtual addresses that seamlessly align with Coyote's native user descriptors.
- The Completion Queue interface is responsible for relaying completion events at the queue pair level to the broader system. It informs when remote writes or sends have been completed upon receiving acknowledgments. In the case of reads, it notifies the users when the RDMA stack has finished processing the responses.
- The Receive Queue interface operates in the opposite direction of the Send Queues. It handles remote write or read requests directed to the local node.
- All RDMA data passes through the Rx and Tx lines, which are multiplexed for all the RDMA verbs, including RDMA READ, RDMA WRITE, and SEND.
- The Connection Management interface is utilized for initializing queue pairs. This interface is where all the network-level routing information is supplied.

4.4.2 Integration of RDMA

Another crucial aspect of the RDMA stack, particularly in a modern data center environment, is the need for reliable communication between nodes. The high likelihood of network congestion underscores the importance of preventing critical packets from being lost. The InfiniBand standard incorporates functionality to ensure this reliability by relying on retransmissions.

In its basic form, this mechanism utilizes acknowledgments to indicate the successful transmission of packets. If an acknowledgment is not received, it could mean either the packet itself

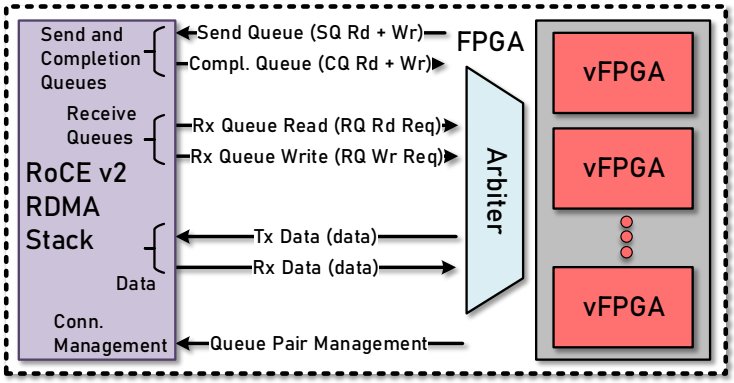

Figure 4.33: RDMA interfaces.

or the acknowledgment signal were lost in transit. In such cases, the sender will retransmit the packets after a certain interval. Packet sequence numbers (PSN) play a crucial role in ensuring the proper handling of packets on the receiving side.

Recognizing the paramount importance of a reliable connection within the Coyote RDMA stack, this functionality is essential. Consequently, the base stack and its retransmission architecture have been adapted to fully support reliable communication.

The primary processing for retransmissions is managed within a dedicated state machine, closely coordinated with a specialized retransmission timer responsible for issuing retransmission events. The retransmission timer is a table where each entry represents a queue pair specific timer. It operates by performing a circular traversal across the table, decrementing each entry. When the timer for a specific QPN expires, it triggers a retransmission event. The timer for each QPN is reset upon receipt of remote acknowledgments.

Additionally, retransmissions can potentially be triggered by explicit NACKs from the remote node, indicating that the packets were not successfully delivered. In such instances, retransmission is initiated without waiting for a timeout. All outstanding requests are retained in a meta table, which includes the original request details that might require retransmission and the associated queue pair information. The state machine of the retransmission operation is shown in Figure 4.34.

It's worth noting that, unlike the TCP/IP stack, which lacks memory abstraction and must buffer all outstanding data, RDMA has the benefit that the original data remains in memory. Therefore, no buffering is necessary, and only the requests need to be preserved for potential retransmission, as the original data can be retrieved from its original location. However, there's a caveat – if the

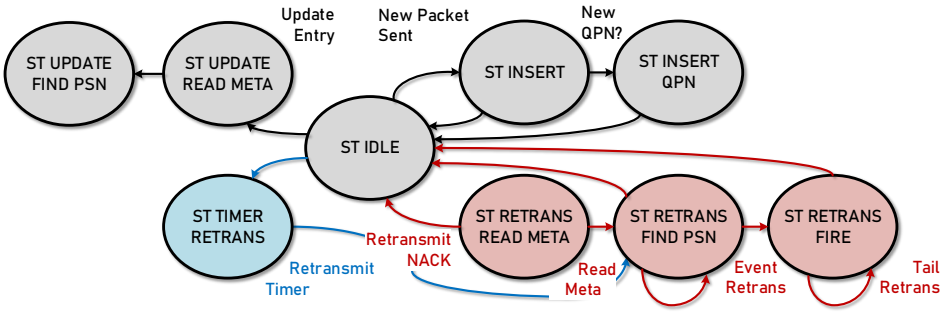

Figure 4.34: Retransmission state machine.

original data is modified in the meantime, it may become corrupted. Similarly, if the original data was fetched from host memory, retransmission could be relatively costly due to the additional round trip over the host interconnect.

To address these concerns, in Coyote, we implement buffering on a queue-pair basis for each of the requests. This means that in-flight data is consistently stored in the FPGA-side memory, effectively mitigating the aforementioned issues. The trade-off is increased resource utilization and a requirement for a dedicated memory channel.

Another critical aspect of the design involves the synchronization mechanisms between remote nodes, necessitating a dedicated flow control system. Without proper flow control, a scenario may arise where the sender can transmit data at a much faster rate than the receiving side can manage. Without a proper backpressuring mechanism across the network, this situation can result in a substantial number of lost packets, wasted network bandwidth, and numerous unnecessary retransmissions.

In Coyote, we adopt a traditional RDMA concept that employs Send Queues and Completion Queues to enforce flow control. What's innovative in this context is that we offload these data structures to the hardware itself. Each Queue Pair in Coyote has the capability to send a specific number of outstanding transactions (compile time parameter). To manage the flow control, we implement a circular buffer in on-chip memory for each of the Queue Pairs. These buffers store minimal information, mainly consisting of pointers. As a result, they utilize a minimal amount of on-chip FPGA memory resources. They efficiently serve as the required backpressuring mechanism for each tenant when the need arises.

Finally, the arbitration between tenants is managed by linking the Queue Pair Numbers (QPNs) to the Coyote IDs. This mapping is illustrated in Figure 4.36. Each QPN consists of 24 bits, within which we encapsulate information about the target tenant and the thread that owns the

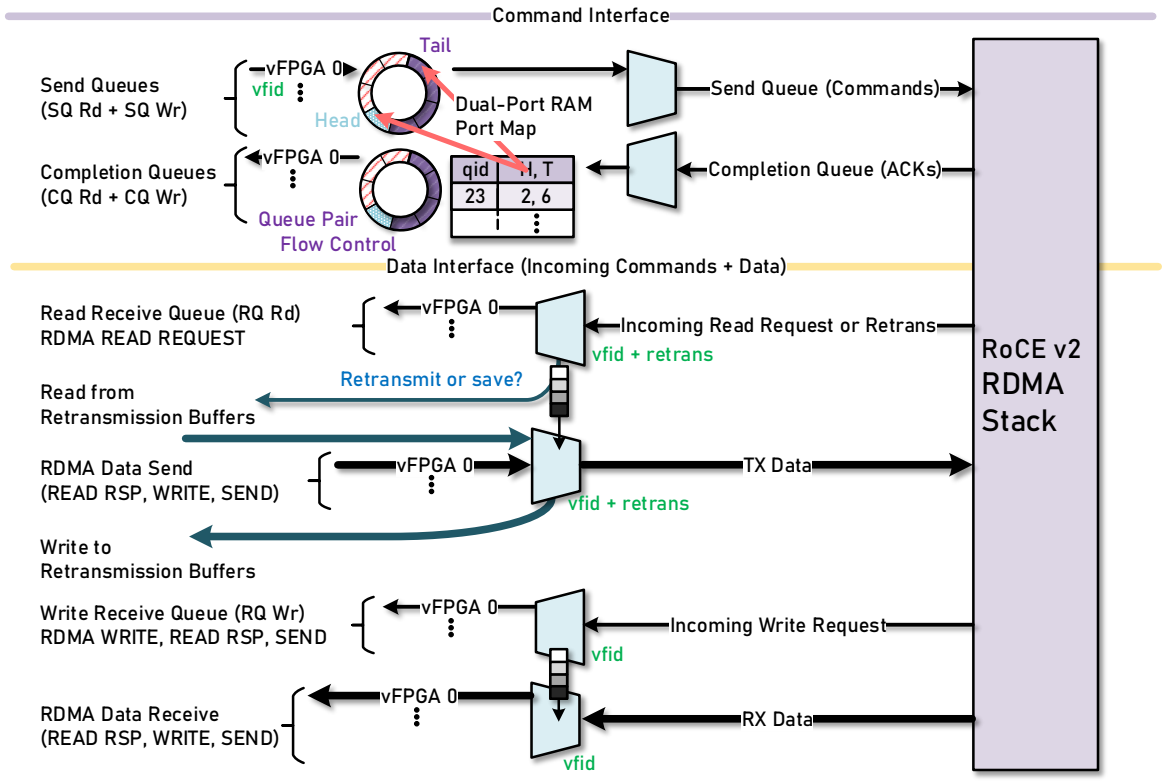

Figure 4.35: RDMA flow control, interleaving and arbitration.

QPN. The remaining portion of the QPN accommodates a session ID, which can be assigned as needed by end users.

Given that the QPN is present throughout every stage of RDMA processing, this design allows us to efficiently integrate this crucial user information not only into requests but also into accompanying data streams where necessary. This approach simplifies the management of multi-tenancy and quality of service, making it a relatively straightforward task. The size of the ID fields in Coyote is a compile-time parameter, making it adaptable to various applications to ensure a sufficient number of queue pairs and supported connections.

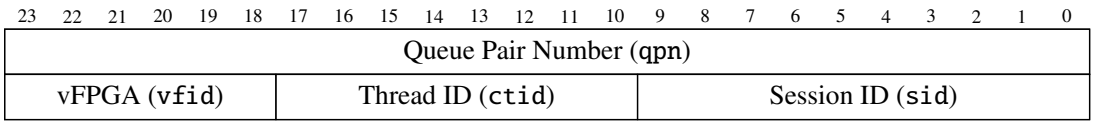

Figure 4.36: QPN mapping in Coyote

4.4.3 Resource Usage of RDMA Stack

The RDMA stack significantly enhances the capabilities beyond those of the TCP/IP stack. Operating at a virtualized level, it offers a robust networking abstraction.

Interestingly however, the resource usage of the RDMA stack is comparable to that of the TCP/IP stack, as shown in Table 4.7. The comparable resource usage can be attributed to the more complex state-keeping structures within the TCP/IP stack when compared to RoCE, which is built on top of a much lighter-weight UDP. These complexities offset the more intricate abstraction of the RoCE protocol.

The utilization of CLBs is quite similar to that of TCP, accounting for approximately 20% of the total resources on the Alveo u55c chip. Block RAM usage is highly dependent on the specific configuration. By default, it is set up to support up to 1000 concurrent Queue Pairs. The trade-off here is akin to the one with TCP. These abstractions come at a cost, which can be mitigated by reducing the number of supported connections.

Additionally, in the case of RDMA, there is an option to disable retransmission buffering and instead use the host memory as the fallback (initial memory locations from which the data was fetched). This not only reduces resource consumption but can also simplify floorplanning tasks, potentially offering benefits even if there are performance and data consistency trade-offs.

Retransmission can also be completely disabled, and the protocol can revert to an unreliable connection (an option supported within RoCE) to further reduce resource overhead. However, it's improbable that this feature will see widespread usage, given that reliable communication holds utmost significance in modern computing environments, as confirmed by our own experience.

Although there are some resource usage penalties associated with it, the enhanced functionality provided by the RDMA stack justifies these costs. However, the question that remains is whether hardening certain portions of these network stacks might be a feasible option going forward. Both RDMA and TCP/IP share lower layers of the stack (such as IP layer). Efficient hardening of these layers, along with retaining the upper layers in the FPGA fabric, can alleviate resource constraints without sacrificing flexibility.

Table 4.7: RoCE v2 RDMA stack resource usage (Alveo-u55c).

Module	CLBs	LUTs	Registers	Block RAM
RDMA stack	31 434 (19.29 %)	167 783 (12.87 %)	213 803 (8.2 %)	150 (7.48 %)

4.4.4 Evaluation of RDMA

We will now assess the performance of the RDMA stack in Coyote, with a specific emphasis on comparing it to commercial RDMA NICs. The experimental setup remains consistent with the previous configuration, featuring a cluster of 10 Alveo-u55c FPGAs. Additionally, we introduce the Mellanox ConnectX-5 RDMA NICs [189] as a reference point for baseline results.

In each experiment, we measure the elapsed time until the conclusive results are stored in the memory of the client machine for both Coyote and the baseline system. To evaluate the system's performance, we assess the network throughput and response time for both RDMA read and write operations. For accurate read measurements, we calculate the network Round Trip Time (RTT) and average it over 1000 iterations. To obtain precise write measurements, we employ a ping-pong exchange between the server and the client. The procedure entails the client first transmitting the writes to the server, and only once these operations are finalized, the server starts similar transfers in the opposite direction. This methodology guarantees that data has been correctly written into the memories, rather than relying solely on ACKs, which may only signify the completion of operations within the NIC.

The transfer size denotes the cumulative data transmitted over the network in a single request. We adjust this parameter until we reach network saturation at least. At the same time we are employing a 4 kB MTU packet size.

4.4.4.1 Comparison with Commercial RDMA NIC (Host Memory)

We will begin with throughput microbenchmarks to establish a comparison between what is commercially available and our implementation. The test results, conducted across various transfer sizes, are depicted in Figure 4.37.

The results demonstrate that the RDMA stack in Coyote consistently delivers more stable outcomes. This is especially evident in the saturation region, occurring at larger transfer sizes, where Coyote attains notably higher peak throughput.

The latency results are presented in Figure 4.38. The difference in latency is not considerable. Our RDMA stack achieves low round-trip times for small transfer sizes, approximately around 6 microseconds, whereas the corresponding latency on the Mellanox card is approximately 4.5 microseconds. While the Mellanox card is a full-blown ASIC with much higher internal clock speeds, it only marginally outperforms our implementation at low transfer sizes. However, its

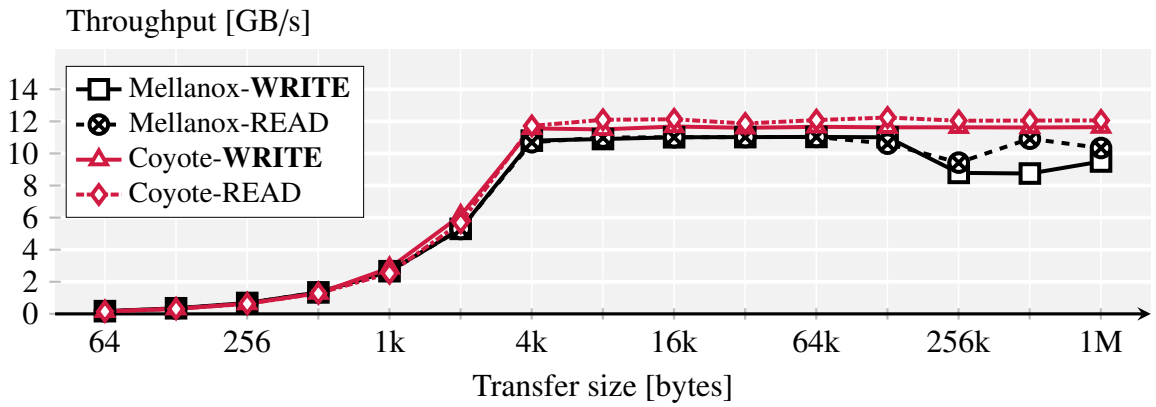

Figure 4.37: RDMA throughput comparison.

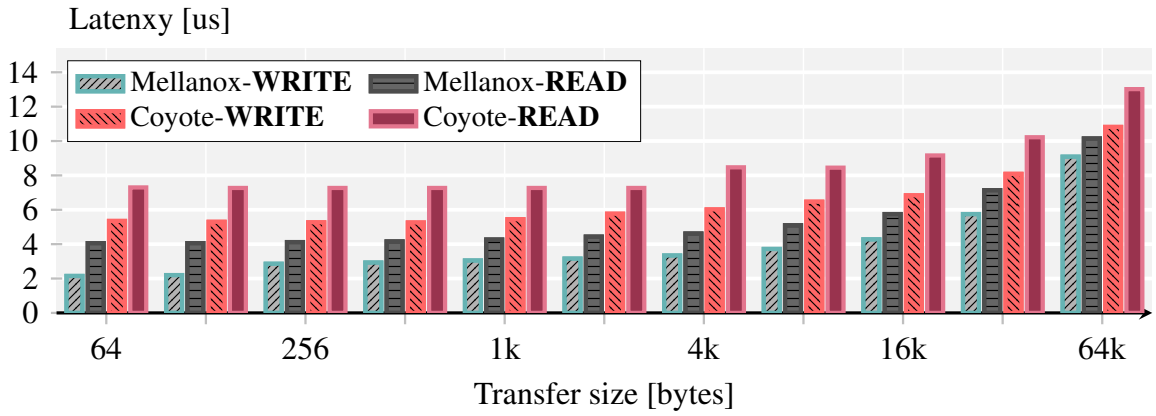

Figure 4.38: RDMA latency comparison.

advantage starts diminishing for higher transfer sizes, where the peak throughput of the FPGA stack comes into play.

These results show that the implementation of the RoCE RDMA stack in the FPGA fabric achieves comparable and in some cases, even better results than a dedicated commercial ASIC RDMA NIC. This is promising, as the main advantage that the FPGA RDMA stack has is not even pure performance, it's the ability to freely pipeline additional computation on top of these RDMA NICs in the FPGA fabric. This is what opens the door for a wide array of use cases and where the main benefit of this abstraction comes.

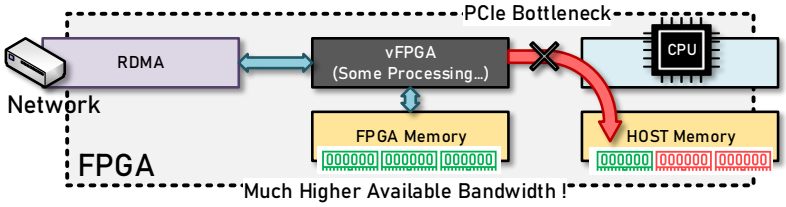

Figure 4.39: RDMA access to FPGA-side memory.

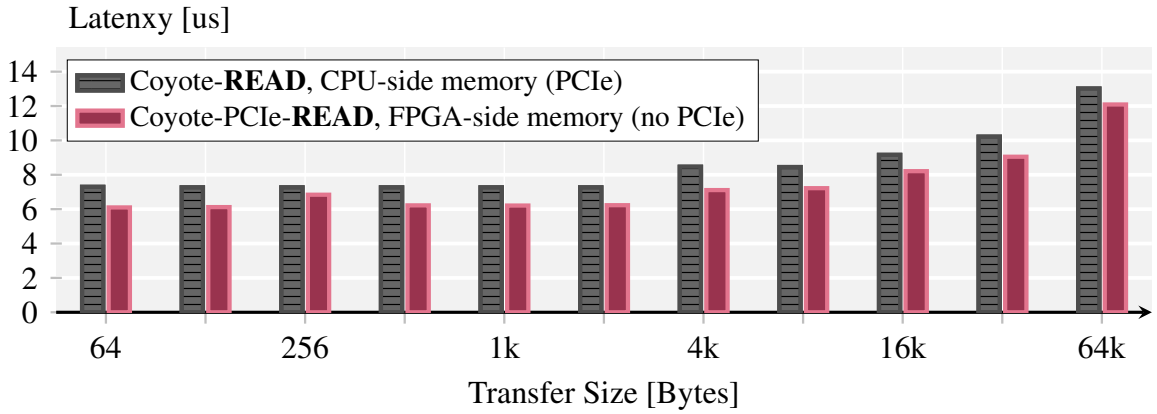

Figure 4.40: Comparison of RDMA Latency: Host Memory vs. FPGA-Side Memory.

4.4.4.2 Host Memory vs. FPGA Memory

In this next experiment, we compare RDMA in Coyote for: 1) data transfer to the host memory (with PCIe in between) and 2) the FPGA-side memory. Notably, we observe that the latency (Figure 4.40) of accessing FPGA-side memory is smaller, approaching the latency of the Mellanox NIC. This outcome served as the primary motivation for investigating smart disaggregated memory architectures, a capability enabled by this setup and explored in a later chapter.

In this scenario where FPGA-side memory is employed, the bandwidth limitation lies in the network (no PCIe). Increased bandwidth of FPGA-side memory can thus be effectively utilized. Imagine a scenario where data filtering is applied, and large portions of data are dropped without being forwarded to the network (e.g. database selection). In this scenario, filtering can be performed at much higher rates than the available network bandwidth, leading to more efficient overall utilization. This stands in contrast to the baseline, where a PCIe bottleneck in the path would limit filtering throughput and ultimately result in lower network bandwidth utilization.

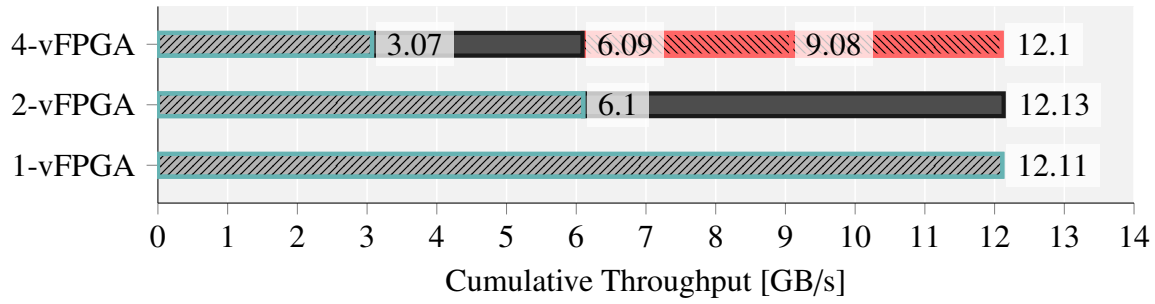

Figure 4.41: Bandwidth distribution when multiple tenants are utilizing the RDMA.

4.4.4.3 Network Multi-Tenancy

Finally, it is crucial to validate the quality of service in Coyote, particularly concerning networking. Given that Coyote is a multi-tenant environment, guaranteeing fair service to each tenant is imperative. Therefore, we conduct an experiment similar to the one we performed for local memory accesses, aiming to demonstrate fair resource sharing when multiple tenants share the same FPGA. The results of this test, conducted with four concurrent tenants in Coyote, are depicted in Figure 4.40. In this case, continuous 1 MB RDMA reads are issued concurrently by each tenant.

As evident from the results, we observe a similar behavior to what was recorded for normal local, non-network accesses from the FPGA. The system maintains fair-sharing across all tenants with minimal variances. This consistency is of great importance as we delve into user logic interfaces in the next chapter. It ensures that the expected behavior of the system remains consistent, regardless of whether accesses are made to local or remote nodes.

4.4.5 Is RDMA a Fitting Abstraction?

Having established the fundamentals of the RDMA stack, we can now elevate the level of discussion and explore how well RDMA fits as an overarching networking abstraction for these heterogeneous platforms.

RDMA, at its core, serves as a memory abstraction. It establishes a connection between two remote machines and facilitates seamless read and write operations. However, heterogeneous systems we are analyzing offer more than just basic memory interfaces. Instead of solely transferring data directly to/from host memory through the network, the primary advantage they offer

is the capacity to perform additional processing on this data. In these heterogeneous system, the ultimate target for network packets is not merely memory; we must also consider the presence of custom processors along the path.

For instance, consider a scenario where some form of processing is applied to the data as it is read from memory. This processing could involve a regular selection operation in response to an incoming data query. In this case, we need the ability to instruct our accelerator along the path regarding what actions to take and what kind of selection to apply. Traditional one-sided RDMA READ falls short in such cases, as it assumes the presence of memory on the other end, rather than a specialized accelerator with a custom interface. Therefore, to fully leverage these platforms, a new approach is required. Otherwise, multiple round trips would be necessary to achieve the desired operation, resulting in significant overheads.

Consequently, what are the prerequisites for the operational model we aim to establish, allowing us to engage these accelerators and, in the end, deliver a suitable framework for our system?

1. **Utilization of Existing InfiniBand Verbs**: We could consider adding new InfiniBand verbs tailored to our needs. However, this approach has drawbacks, including potential incompatibility with commercial NICs, which would hinder interoperability and complicate integration into modern data centers. As a result, our space for exploration is limited to the existing verbs within the protocol.

2. **Targeting Traditional One-Sided Verb Performance**: We aim to achieve the performance levels associated with traditional one-sided verbs. Introducing additional round-trips or significantly increasing latency would diminish any advantages.

3. **Diverse Accelerators on the Path**: Given the FPGA's versatility, it can accommodate a range of accelerators with distinct behaviors. Consequently, we require a generic method for invoking these accelerators. Ideally, users should be equipped with a universal interface for this purpose.

4.4.5.1 Combination of One-Sided and Two-Sided Verbs

The solution that appears to satisfy all the necessary requirements is to utilize a combination of one-sided verbs and traditional two-sided verbs. In this approach, instead of using reads to invoke our remote accelerators, we achieve the same outcome with a two-sided SEND verb, copuled with one-sided RDMA WRITE in the other direction.

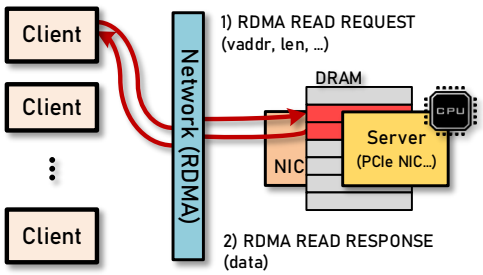

Figure 4.42: Traditional one-sided RDMA verbs.

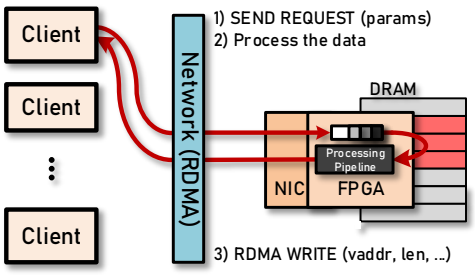

Figure 4.43: Combination of one-sided and two-sided verbs.

In traditional terms, two-sided verbs involve the CPU handling all the received data. However, this behavior doesn't align with our goals. We utilize the RDMA stack precisely to avoid using the remote CPU and wasting its computational resources. In our approach, we employ the semantics of the SEND verb, but the response to this command doesn't come from the remote CPU. Instead, it's the FPGA fabric that takes action. In essence, although we use two-sided verbs, we are employing them in a one-sided manner, avoiding the consumption of CPU cycles.

These SEND verbs are also highly versatile, and can encapsulate various types of accelerator invocations. For example, if we revisit the selection example, we can include multiple predicates within this verb, which will be passed to the accelerator directly. Simultaneously, we can issue an instruction to read the data from memory, which will then undergo filtering in the accelerator. Finally, the results produced by the accelerator are sent back over the network using regular RDMA WRITE verbs, completing the round trip. This approach allows us to utilize the supported set of verbs without adding significant overhead to the critical path. Moreover, it provides a generic interface as SEND verbs can carry arbitrary data of varying sizes.

While this approach meets the specified requirements, it does come with certain drawbacks. For instance, it necessitates the inclusion of detailed buffer information within the SEND verb since there are no dedicated headers that convey this data. Sharing such information with the remote machine could potentially introduce security vulnerabilities. It may also not be the most suitable abstraction for our specific needs. In fact, RDMA itself might not be the ideal abstraction for the current landscape of heterogeneous architectures and the increasing hardware complexity in modern computing systems. Much of the complexity in RDMA could be considered unnecessary, and future exploration could lead to the development of a lighter-weight generic protocol built on top of UDP or similar technologies.

4.5 Summary

In this chapter, we presented an in-depth overview of services that are provided to the end users within Coyote.

We first explored Coyote's memory organization, beginning with the physical layer and working our way up to the high-level shared virtual memory model.

Additionally, we presented a view of the critical networking services, which often serve as the primary source of performance advantages provided by FPGAs.

In the following chapter, we will start exploring the user application layer. We will examine how Coyote establishes an environment for running untrusted applications and how these applications can ultimately interact with the system.

Chapter 5

Application Layer: The Virtual FPGA

After we've covered the crucial system architecture and the services in the system, we can shift the focus to the application layer.

The application layer consists of multiple parallel vFPGAs, which, as mentioned previously, closely resemble a conventional concept of an operating system process. Similar to processes, they play a crucial role in establishing a standardized execution environment, guaranteeing that an application running in one process will also run consistently in another. They can also be time-shared, resembling the concept of a context switch to a certain extent, albeit without any preemptions involved. The vFPGAs are split into two sub-layers. The first is the user wrapper, and the second is the actual user application.

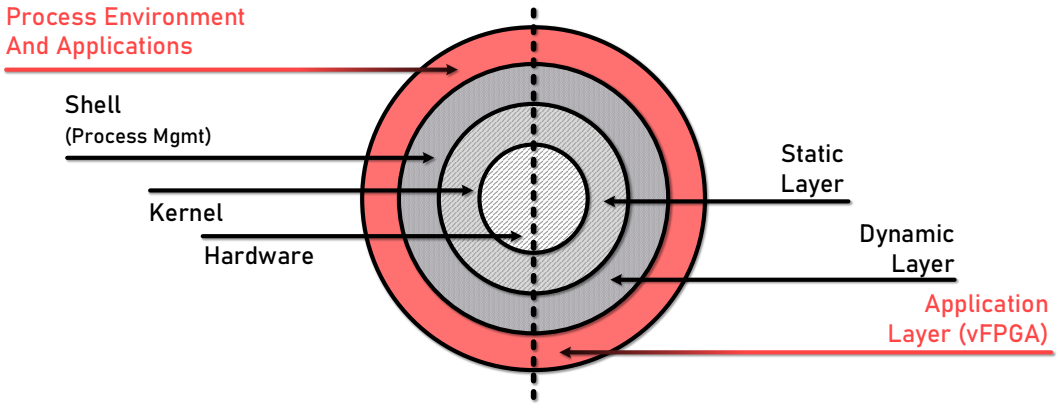

Figure 5.1: Application layer

Since Coyote is a virtualization framework, the concept of the user wrapper closely resembles that of a virtual machine monitor, commonly encountered in modern cloud environments. This is typically either a software or a hardware layer that creates and manages the virtual machines. It abstracts the underlying physical system resources, ensures the required isolation and provides the ultimate interfaces to virtual machines. In this context, these user wrappers serve as the ultimate layer of the Coyote hypervisor. They encapsulate the applications and protect the overall system from user deployed hardware. Sticking with this virtualization paradigm, the user applications can be compared to the actual virtual machines. What hardware gets deployed within these "containers" is determined by the system's end users.

We will start the chapter by exploring the user wrappers and the hardware interfaces they present to hardware applications. With this, we will conclude our study of the overall hardware infrastructure in Coyote.

Having covered all the hardware abstractions, we will demonstrate how the virtual FPGAs can be made accessible to users. To truly justify the "virtual" aspect of their name, we will illustrate how they can be exposed beyond mere interaction with the host OS. Instead, we'll reveal how traditional hypervisors running on the host OS can elevate these vFPGAs to the level of virtual machines, enabling interaction with them from different virtual operating systems concurrently within the same host.

This will finally enable us to begin our exploration of the user software API in Coyote. When combined with the virtual machine abstraction and all the services featured so far, this will constitute the ultimate abstraction for end users provided by the system.

We will end the chapter by validating the overall framework, presenting examples that demonstrate how the presented abstractions can be applied in real-world practical scenarios, yielding significant performance advantages.

5.1 Unified Logic Interface

For a long time, a fundamental challenge of modern FPGAs was the lack of standardized execution environments. Each FPGA platform arrived with its unique set of constraints and interfaces. Consequently, migrating hardware designs was anything but straightforward and often demanded substantial engineering efforts. This situation resulted in developers dedicating a substantial amount of time to design the necessary infrastructure for their applications to operate in,

rather than focusing on the application development itself. Even after this foundational work was completed, the designs would exclusively function on the specific platform. Porting them to another platform often required repeating the same extensive efforts.

To solve this issue, standard execution environment for AMD Xilinx FPGAs was first provided through SDAccel [7] and subsequently Vitis [207] flows. These environments, coupled with dedicated FPGA shells [208], served as the foundations for the widely adopted cloud FPGA system, the Amazon F1 [30]. Comparable environments were also offered by other vendors, such as Intel's HARP [62].

The drawback of all these environments was the notable shortage of features they provided. They restricted FPGA functionality to a limited set of capabilities, severely constraining the potential of the FPGA and reducing the range of features exposed to the end users. This, in turn, restricted the overall advantages that these devices could potentially offer. It's reasonable to speculate that with a broader range of features, the adoption of these devices could have been more widespread.

In Coyote, we establish a standard execution environment by developing a *Unified Logic Interface* that provides a well-defined set of interfaces accessible to users. Significantly, these interfaces do not impose limitations on the FPGA features, granting access to all the abstractions we have discussed thus far.

5.1.1 Service Interfaces

First, let's examine the interfaces of all the services we've covered so far, shown in Figure 5.2.

5.1.1.1 Control and Notification Interfaces

To manage the operation of the deployed hardware applications, users need to have the capability to control it from the software side. To serve this specific purpose, a dedicated AXI4 Lite bus in hardware is made accessible to the software. This lightweight bus is memory-mapped into the application's user space, allowing users to perform direct read and write operations. Each virtual FPGA has its own distinct address range for this interface. Initially, the driver maps this address range from the I/O space into the kernel's virtual address space. Subsequently, the software API further maps it directly into the users' virtual address space. Importantly, this arrangement bypasses the host kernel, resulting in a reduction of operation latency.

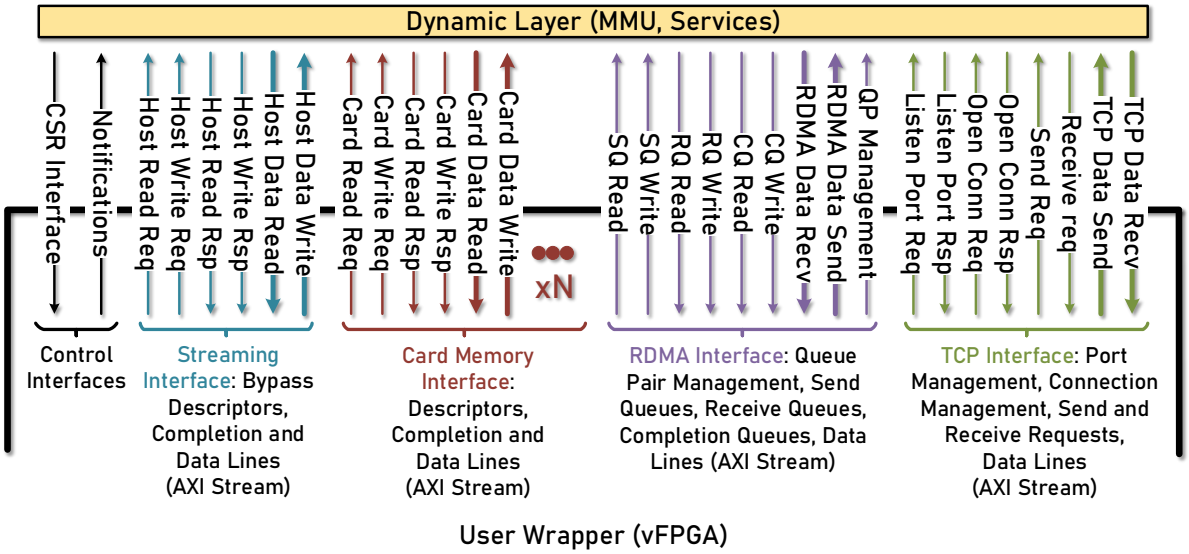

Figure 5.2: Services and Interfaces exposed to user wrappers and user logic in Coyote

Within the hardware, this interface is connected to a collection of control and status registers, and the function of these registers is entirely determined by the specific application and the software API. Through this interface, users have the ability to manage, manipulate and monitor the operation of their hardware.

Listing 5.1: Control Interface memory mapping accross the stack.

```
// Driver code: Map from I/O -> Kernel VA
d->vfpga_dev[i].fpga_cnfg =
    ioremap(d->vfpga_dev[i].fpga_phys_addr_ctrl, FPGA_CTRL_CNFG_SIZE);

// Software API: Map from Kernel VA -> User space VA
ctrl_reg = (__m256i*) mmap(NULL, ctrlRegionSize,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd, mmapCtrl);
```

However, in many scenarios, it's not efficient to continually poll hardware applications to check for specific events (this would consume valuable CPU cycles). A more effective approach is to leverage interrupts. Therefore, Coyote offers a dedicated notification interface accessible to user hardware. This interface can be viewed as the opposite of the previous AXI Lite control bus we discussed. Instead of granting only the host software application the ability to notify the hardware, the hardware application can also inform the software about a particular event.

Through this interface, hardware applications can send aribtrary 32-bit integer values which will be read by the driver during the interrupt handling. These will then be immediately forwarded to the user space of the corresponding software application. To achieve this functionality, an

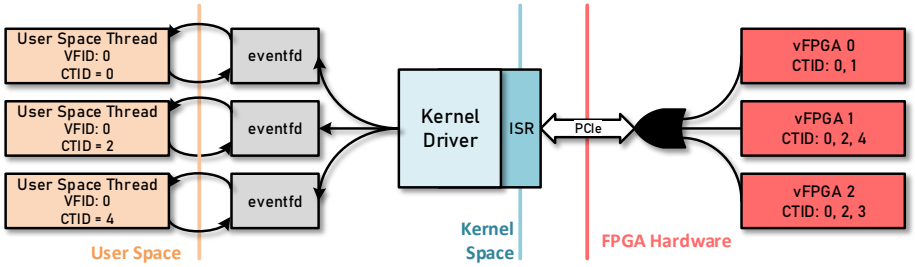

Figure 5.3: Hardware notification mechanism in Coyote

eventfd object is used. The eventfd serves as an event wait/notify mechanism within the kernel for user-space applications. In a similar fashion to traditional signal handlers within the kernel, when the driver receives an interrupt, it utilizes the eventfd interface to asynchronously signal the specific Coyote thread in the user space. A separate user space thread will then periodically check for this signaling event.

The advantage of this approach over traditional signal handlers is that eventfd permits the registration for each Coyote thread, which is not feasible with traditional signal handlers, which can only interrupt the entire process. In that case, if the user has registered multiple Coyote threads, these users would need to additionally determine which one triggered the signal. With eventfd signals are sent with specific thread IDs (*ctid*), and each thread is allocated a distinct handler, making it easier to pinpoint the source of the signal.

5.1.1.2 Local Data Interfaces

To initiate a data transfer, whether it's a read or write operation targeting either local memory or FPGA-side memory, user applications must send a descriptor that includes essential information such as the virtual address of the data location, the length of the transfer, the Coyote thread ID, and a set of flags indicating the requested transfer details. Among these flags is the specification of the target location for the data, which can either be in the host or the FPGA-side memory.

Once the request is initiated, it is ultimately passed on to the memory management unit, which takes charge of processing the request and routing it to the correct destination. The data transfer is then initiated. In the case of reads, data will start arriving through the slave AXI streams, while for writes, users will need to provide the data to the outgoing streams. It's worth noting that all these interfaces, including the descriptors, adhere to standard AXI streams, making integration straightforward, whether you're working with traditional RTL or some form of HLS.

It's important to highlight that there are several parallel data interfaces leading to the FPGA-side memory, crossing over from the user wrappers to the dynamic layer. This is because the FPGA-side memory boasts high bandwidth, and by having these parallel interfaces, the full bandwidth can be effectively utilized. The broader management of bandwidth distribution at a higher level among virtual FPGAs is handled by the MMU and the physical memory layers, as discussed in earlier chapters.

5.1.1.3 Remote Data Interfaces

Much like the set of interfaces for local memory services, a similar set of interfaces exists to interact with all the remote (network) services in Coyote.

The TCP stack in Coyote operates at lower level of abstraction compared to other services. The primary reason for its inclusion and continuous support in Coyote is its widespread use in modern networking, particularly for implementing control plane operations. When it comes to high-performance data transfers, user applications should rely on the RDMA stack.

In essence, TCP stack, apart from carrying raw data payloads, lacks higher-level metadata within its data streams. Unlike all the other interfaces, it doesn't include any notion of memory or the ultimate data destinations. This is in contrast to RDMA, which, while delivering line-rate performance, essentially provides the same interface as all other local data interfaces targeting the memories within the local node.

Hence, the interaction with the TCP stack involves traditional two-sided send and receive operations. When it comes to send operations, only the length, Coyote thread ID, and connection session ID need to be provided (along with the accompanying data). In other direction, the user logic will be notified whenever a remote node sends data. This is done through the receive interface. The meta information remains the same, but in this case, it applies to incoming data. In addition to these data interfaces, the set of TCP interfaces also encompasses connection management interfaces.

The RDMA interface comprises the following essential components: the send queue, the receive queue, and the completion queue. The send queue is responsible for initiating data transfers to a remote node. To do so, users need to provide the same set of information as if they were transferring data to local memory, which includes the virtual address, length, and thread ID. This time an additional session ID is also needed that identifies the connection.

From these signals (Coyote thread ID, session ID, and vFPGA ID), the queue pair number is generated. This request is passed to dedicated circular buffers, from where it is forwarded to the

RDMA stack. The receive queues contain identical information, with the distinction that they signal remote nodes writing to a specific local memory and a particular Coyote thread. Similar to TCP, RDMA also features an interface for managing connections, primarily to register and manage all queue pairs within the system.

5.1.2 Unifying the Interfaces

All these interfaces predominantly rely on AXI streams, making integration a straightforward process. User logic can interact with these interfaces by simply establishing a basic handshake mechanism to transmit or receive data. Example handshakes in RTL and HLS are shown in Listing 5.2.

Listing 5.2: Sending data through the AXI stream (handshake in RTL and HLS).

```
// RTL (Verilog)
host_read_req.valid = 1;
host_read_req.data =  {vaddr, len, ctid, last, ...};
if(host_read_req.ready) {
    // Read request sent
    // ...
}

// HLS
host_read_req.write(rdReq(vaddr, len, ctid, last, ...)); // Blocking
// Read request sent
// ...
```

The emphasis on user-friendliness is crucial, as one of the primary objectives of the framework was to enable the simple integration of custom user circuitry. However, if we take another look at Figure 5.2, which illustrates the interfaces across all services, it becomes apparent that the number of these interfaces is substantial. Although they all conform to these uncomplicated streaming protocols, individually handling each interface can remain challenging due to their sheer quantity. When considering that each of these interfaces is accompanied by its own set of specific data structures, the complexity of integration can escalate quickly. Secondly, the large number of these interfaces also increases the number of overall signals that have to cross from one reconfigurable region to another which has a detrimental impact on overall resource usage and routability of the design.

The solution lies in unifying these interfaces (shown in Figure 5.4). This is where the virtualization abstractions within Coyote become essential. Thanks to virtualization, each data interface in Coyote relies on standard virtual address pointers (except for TCP, which solely requires the

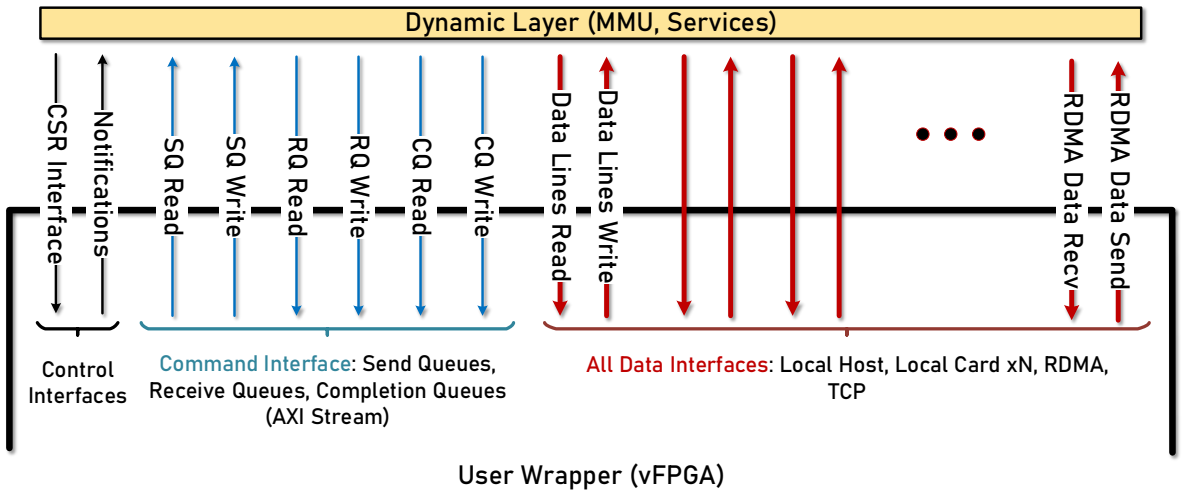

Figure 5.4: Services and Interfaces exposed to user wrappers after unification.

session IDs). Consequently, we can merge a number of these interfaces into a single one. So, the question arises: which type of interface would be suitable for this unification?

As previously discussed, RDMA is constructed upon a memory abstraction, much like local data requests. If we dive into the RDMA framework, it uses the concept of send queues, receive queues, and completion queues, which is a conventional mechanism for data operations (also commonly found in modern storage devices). When we examine the other interfaces in Coyote, they all essentially revolve around the same fundamental concept: the transfer of data from one location to another, primarily through DMA operations. Consequently, this abstraction is appropriate for all the interfaces in Coyote, enabling us to consolidate them under a single set of send, receive, and completion queues. Any instruction that the user application needs to issue can be executed through this unified interface, whether the data is local or remote.

To further remove unnecessary interfaces, we can also relocate network management operations and control them through slave registers via AXI Lite buses, directly from the CPU. We can do this as they do not affect the critical path of user applications in any way. This further simplifies the system. Consequently, apart from the send, receive, and completion queues, we are left only with regular data lines, greatly simplifying interaction and enhancing ease of use.

Despite a notable reduction in the overall number of interfaces, a question arises regarding how this streamlined set of interfaces can effectively cover the entire spectrum of services in Coyote. We can demonstrate this with the example of the primary invocation interface, the send queues (Figure 5.5, two per vFPGA, one for reads, one for writes).

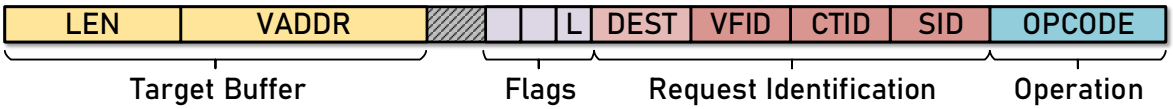

Figure 5.5: Send queue data structure (unified request).

Table 5.1: Send Queue (SQ) interface

Field	Description
`Opcode`	This field determines both the nature of the operation (read or write) and whether it is executed locally (HOST or CARD) or remotely (RDMA or TCP).
`Ctid`	Coyote thread ID assigned to this request. Any data corresponding to this request will have the same ID.
`Vfid`	vFPGA ID. This is a reserved field. It is tied to the correct value in the dynamic layer.
`Sid`	Session ID. Used for network requests to identify the connection.
`Dest`	Destination field. It is used to designate the target queue within the user wrapper for the data. This field will be covered in the following section.
`Last`	Last transfer, trigger a completion event.
`Vaddr`	Virtual address. The address of the buffer from where the data is read or where it will be written. Importantly, this can be either local address or a remote one.
`Len`	Length of the transfer.

The receive queues (RQ) share an identical data structure with the send queues, with the distinction that they handle incoming data and serve to notify the hardware application when a remote node initiates a network request, either for data retrieval or data writing to a local destination. Finally, the completion queues (CQ) are responsible for informing hardware applications about the completion status of specific send queues.

With this streamlined set of interfaces, user applications can easily interact with the full array of services, whether they are local or remote. This simplification of the overall hardware interface significantly boosts design productivity.

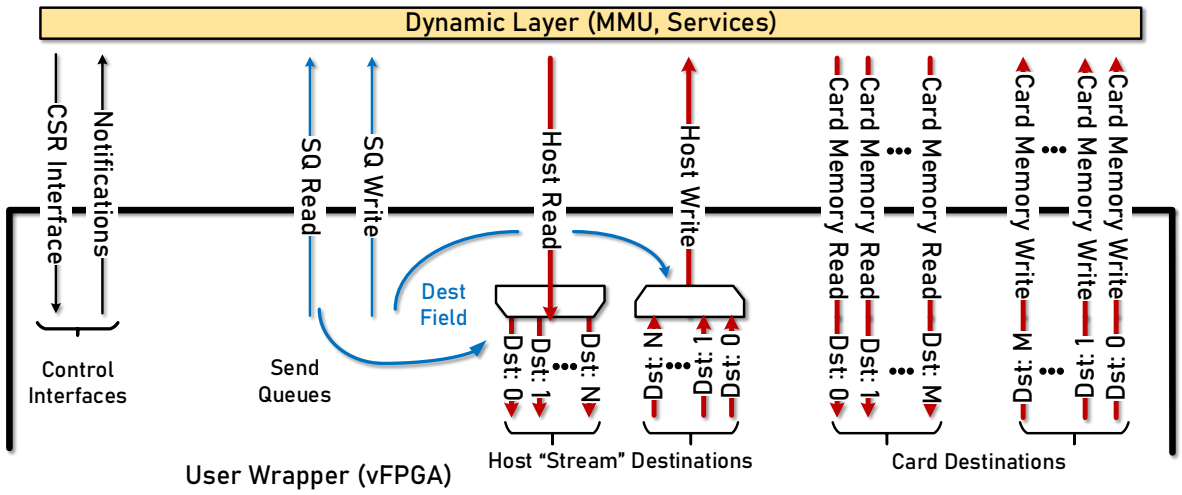

Figure 5.6: Parallel destinations within the vFPGA.

5.1.3 Parallel Data Streams

Before we begin examining how we provide the untrusted container to the end user applications, we will present an additional interface abstraction within the user wrapper. This abstraction primarily focuses on simplifying the integration of user applications, especially when there is a need for multiple simultaneous streaming interfaces from different sources.

In the initial versions of Coyote, we had only one set of interfaces for each service in the system. This meant that each vFPGA had a single set of slave and master data lines (streams) to the host memory, the card memory and network services. The reason for this design choice was that introducing additional data streams wouldn't necessarily result in performance improvements, as a single set was sufficient to saturate the available bandwidth.

However, this situation differed for the FPGA-side memory. Despite the bandwidth being shared among multiple vFPGAs, scenarios could arise where there is unused bandwidth left (HBM can support up to 400 GB/s). In such scenarios, it became advantageous to establish parallel channels to this memory within a single vFPGA. This is where the *dest* field comes into play. It designates the queue in the user wrapper, which will either be used to store data (in the case of a read operations) or from which data for writing will be retrieved (in the case of write operations). This allows user applications yet another interface to organize and distinguish their data. Although this functionality was initially introduced to make the most of the available FPGA off-chip memory bandwidth, it was extended to other interfaces as it proved quite useful.

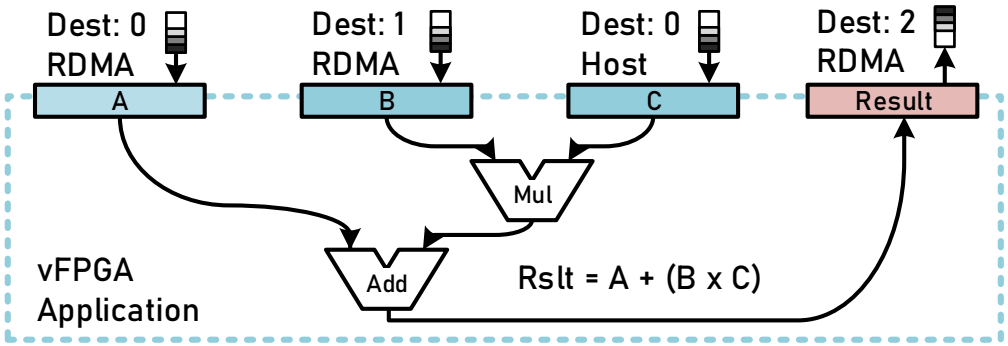

Figure 5.7: Example of user application (multiply-add unit) which utilizes multiple parallel streams.

An illustrative example of a potential application for these parallel streams is depicted in Figure 5.7. In this scenario, a multiply-add unit can be seamlessly integrated into the user wrapper as parallel streams allow for the easy retrieval of multiple inputs from various sources (without any extra logic needed from the application). The only task required of the user application is to issue multiple requests with the *dest* field set to the appropriate queue.

Significantly, as these requests are further broken down into packets and interleaved at lower levels of the stack, users can be assured that potential head-of-line blocking issues will not arise. This concern is effectively managed by the system through interleaving and a crediting system, which we will cover in the next section.

5.1.4 Untrusted Environment

An important aspect of vFPGAs is their ability to accommodate arbitrary user logic, as long as it conforms to the provided interfaces. Drawing a parallel with operating system processes, one key consideration is that applications running within them are inherently untrusted. Therefore, it's essential to take all necessary precautions to protect the rest of the system from malevolent behaviour. This is why software applications can only access memory locations that the kernel explicitly grants them access to. In Coyote, similar protection is ensured by employing a dedicated memory management unit to enhance the security of the entire system in this context.

However, when dealing with hardware, additional protective measures must be taken into consideration. In hardware applications, operations occur at a much lower level, involving the manipulation of electrical signals that propagate throughout the system. This implies that, in ad-

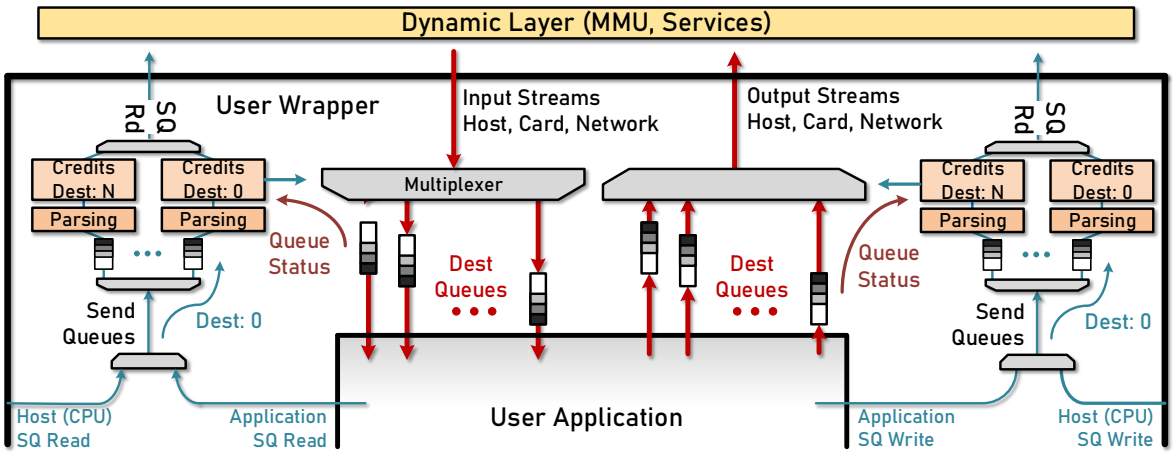

Figure 5.8: Credit based system.

dition to ensuring the security of memory, we must also safeguard other concurrently operating hardware modules.

As an example, let's imagine a situation in which a single vFPGA requests data but, upon receiving the data, fails to consume it. In this case, this single vFPGA (process) has the potential to create backpressure on the entire system, thereby disrupting the operation of other vFPGAs and the system as a whole. Similarly, the reverse situation could occur, where a request is made to write some data, but no data is subsequently provided. In such a case, the pipelines throughout the system could potentially stall.

Consider the impact if processes in a traditional operating system were capable of causing these problems, it would lead to significant issues to say the least. Therefore, a solution is required to preempt these potential issues and effectively isolate user logic from the overall system.

5.1.4.1 Credit System

In Coyote, the solution used is a credit-based system, which is implemented in the user wrapper of each vFPGA (Figure 5.8). This credit-based system is built on top of the destination queues discussed in the previous section.

The operation is as follows: when a request is initiated through a send queue interface, it is first directed to the appropriate credit module, which is separate for reads and writes. It's worth mentioning here that the send queue requests are multiplexed prior to this, combining both host-issued (software application) and FPGA-issued (hardware application) requests. This is how

Coyote enables both host-CPU and FPGA-side initiated requests. This credit module then breaks down the request into packet-sized chunks, which are then passed to the module responsible for verifying whether there are enough credits available to send this request to the MMU. These credits are tied to the current usage of the destination queues in both directions. The number of outstdanding requests that can be issued for each queue is a compile time parameter.

To illustrate this process, if a read request is generated, the credit module evaluates whether there is sufficient space in the target receive queue (based on *dest*) for a read packet. If there isn't enough space, the request won't be forwarded, and backpressure will be applied to the user logic rather than the system below. The same principle applies to write requests, where the request is only forwarded if enough data (the required packets) has been provided to fulfill the entire request. This approach ensures that the operation of a single vFPGA does not adversely impact the rest of the system in the dynamic layer or other hardware processes.

This solution is employed for all outgoing requests, regardless of whether they are local or remote. However, because networking operations typically involve significantly higher latencies, the receive and send destination queues for these operations are sized more generously, accommodating a larger number of possible outstanding transactions. This approach assists in hiding the inherent latency of networking operations.

Furthermore, the network receive queues exhibit different behavior (Figure 5.9) since, in this case, the application is not the one initiating the request. Instead, requests are incoming over the network (through which there is no exchange of credit information). Consequently, we must consider adversarial behaviour for this interface as well. When the application receives requests but does not consume the data, it can potentially backpressure the entire system and, ultimately, the network stacks. This situation could lead to the potential stalling of all other network packets that are received. To address this, we allow the network stack to probe the receive queues directly and check if they have adequate space for incoming packets. If there is sufficient space, the packet will be forwarded. However, if the receive queue lacks space, the network stack itself will promptly drop the packet and fall back on retransmission.

In this manner, if the packets are not yet ready to be received by a specific tenant, it won't negatively impact other tenants, and the packets will be received at a later time when space becomes available.

Lastly, if the hardware application fails to cooperate for an extended period, and given the absence of preemption mechanisms, it will be terminated. The advantage of having the credit system within the dynamic region (in the user wrapper of the application) is that during removal,

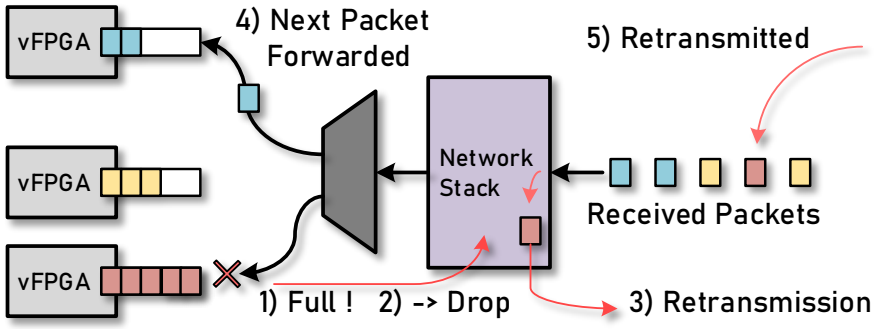

Figure 5.9: Dropping receive packets.

any potentially remaining internal state is cleared. This ensures that the next application can be loaded safely, and once the loading process is completed, it's immediately ready for operation.

5.2 Virtual Machines

Virtual machines (VMs) are software emulations of the underlying hardware, enabling multiple environments, such as operating systems and applications, to operate on a single piece of hardware. Each virtual machine operates independently, with its own isolated set of resources and configurations. This isolation helps protect against conflicts, security breaches, and potential malicious actions in one virtual machine affecting others or the underlying physical hardware. Consequently, virtual machines serve as the primary means for renting modern FPGAs (among other devices) in cloud environments. [30, 26].

However, the current cloud rental model involves virtualizing the entire FPGA as a single PCIe-attached device. In Coyote, our aim is to take a different approach by exploiting the hardware multi-tenancy in the framework. In this approach, individual virtual machines are allocated for each distinct virtual FPGA (and its untrusted application). Thus, instead of allocating an entire FPGA to a single user and launching a single virtual machine, we are able to offer simultaneous access to multiple users, each running in their separate virtual machine.

This final layer of virtualization in the Coyote framework (covered in [74]), utilizing virtual machines, aligns well with the core principles of multi-tenancy and user isolation. This combination of hardware and software based emulation enables much more efficient utilization of the modern cloud resources.

5.2.1 Virtualization of the Host Resources

Before we can proceed with the virtualization of FPGA hardware, the initial step involves assessing how to prepare and virtualize the existing CPU resources on the host side. Modern computing offers a wide range of virtualization mechanisms, encompassing bare-metal hypervisors, OS hosted hypervisors, even extending to full user-space emulation. In our case, we decide to opt for a traditional way things get virtualized in Linux kernel, by using an open-source Kernel-based Virtual Machine (KVM) [174].

This is a virtualization technology that transforms the underlying operating system into a Type-1 (bare-metal) hypervisor, running directly on the physical hardware. Its close integration with the Linux kernel makes it a suitable choice for Coyote, especially since the primary management layer within our system relies on the Linux kernel as well. It also comes with low requirements since the majority of modern physical hardware supports this type of virtualization (Intel VT-x [194] and AMD-V [2]). This hardware support primarily offers hardware-based nested paging capabilities that greatly improve the efficiency of translating memory addresses between guest virtual machines and the physical hardware. These hierarchical tables manage the resource-intensive translations from guest virtual to host virtual addresses and then from host virtual to physical addresses. Additionally, they are the enabling factor for running multiple VMs on a single physical host while preserving their distinct address spaces. If one VM attempts to access memory beyond its allocated range, the nested page table guarantees that it remains isolated from the memory of other VMs or the hypervisor.

Besides the CPU and memory resources, virtualization of other hardware components, such as GPUs, NICs, FPGAs, etc., is not handled by KVM and is typically the responsibility of the user, who may use additional tools to implement this virtualization. One such tool is QEMU [52]. QEMU is an open-source emulator that goes beyond emulating traditional resources like the CPU and memory; it also targets various other hardware components. Notably, it includes support for emulating a PCIe bus and the devices connected to it. This PCIe bus virtualization enables us to virtualize the FPGAs and the entire hardware deployed within it. Therefore, QEMU is a preferred tool for achieving the virtualization of a heterogeneous platform.

The QEMU offers a software-based mechanism for users to manage emulated devices. When a virtual machine operating within QEMU sends I/O requests to a specific device, QEMU intercepts these requests and emulates the expected behavior of the device using software routines. This emulation can be entirely executed in the user space. Notably, QEMU also supports complete pass-through of hardware devices.

Access from the user space is done by mapping the address range exposed by the physical device directly into the virtual machine's memory space. When users access this memory, it does not trigger a trap, and the access is directly forwarded to the device. While this method may raise security concerns, particularly for generic devices shared among VMs, it is not a concern in Coyote. Coyote's lower layers of the stack are designed to handle multi-tenancy, ensuring secure interaction with devices from VMs with virtually no performance overhead.

Generally speaking, QEMU offers a notable level of flexibility when it comes to determining the extent of virtualization applied to a specific device. This flexibility enables users to customize device virtualization to meet their particular needs and preferences, as is the case in Coyote.

5.2.2 Background

Modern FPGAs typically lack extensive built-in virtualization hardware support and often rely on standard DMA operations over PCIe, that work with physical addresses only. However, efforts have been made by companies like AMD to introduce specialized cores (QDMA [35]) that incorporate virtualization support directly into the hardware. This hardware support is based on the Single Root I/O Virtualization (SR-IOV) technology [4].

SR-IOV is primarily aimed at enhancing the performance of virtual machines (VMs) by offloading critical components of the virtualization framework to the hardware. The hardware device can then partition its internal resources to accommodate multiple concurrent VMs (referred to as virtual functions), along with the necessary management layers for the host hypervisor (referred to as physical functions). The hypervisor can then control the device's operation directly (without traps) through the exposed physical functions in the host kernel.

However, it's important to highlight that SR-IOV support is exclusively available for the QDMA core and is not extended to other PCIe cores provided by AMD. This core unfortunately comes with certain limitations and a strict operating model for the host driver (based around ring buffers). As a result, it provides less flexibility and involves a more complex integration process when compared to XDMA, used in Coyote. The virtualization hardware integrated within the FPGA would also utilize a substantial amount of extra resources, diverting them from other logic. In addition, the vendor FPGA shells that included this core were discontinued, and there may be a lack of future support.

All these factors make it a less viable choice for virtualization in Coyote. As a result, our system relies on software emulators to achieve system-wide virtualization, ultimately providing similar functionality without these hardware limitations.

It's important to mention previous work in the field of multi-tenant FPGA hypervisors. One such example is the Optimus hypervisor, which was created to virtualize multi-tenancy in FPGAs on the Intel HARP platform [62]. We discussed Optimus already in the background section of the Chapter 2.

Optimus operates in a way similar to Coyote, but there's a key difference. In Optimus, the design is fixed, and it doesn't support dynamic changes. Instead, a set number of accelerators (basically vFPGAs) are programmed into the FPGA, and these can then be arbitrarily assigned to different virtual machines running on the host hypervisor.

Optimus also supports the shared virtual memory model, but with a limitation: it doesn't consider FPGA-memory resources. The way this shared memory model is implemented differs significantly from Coyote. In Optimus, the system relies on the host IOMMU rather than having a dedicated MMU within the FPGA fabric.

To make this memory model work for multiple tenants and VMs on top, Optimus divides the page table into multiple regions. Each region is allocated a distinct address space, corresponding to a different tenant. To ensure proper address translation, all virtual addresses are given an offset. When a user's logic accesses data using an address, the FPGA shell adds this offset to the address. This adjustment ensures that the translation occurs within the correct region of the page table.

One significant distinction, apart from the use of a host IOMMU, is that Optimus lacks the capability to handle page faults. In other words, if the user's logic encounters a fault, there are no built-in mechanisms to address and resolve that fault. As a result, all mappings must be explicitly managed by the user space program. This approach somewhat simplifies the virtualization and the implementation of the hypervisor.

Crucially, Optimus provided us with the concept of implementing and operating virtual machines on individual vFPGA devices within a single FPGA. This concept primarily involves the utilization of the Virtual Function I/O (VFIO) Linux kernel API to virtualize device files in the kernel, coupled with KVM for virtualizing the host CPU and memory resources.

5.2.3 Virutalization of vFPGAs

The hypervisor is constructed on top of the existing Coyote device driver, extending it to support multiple virtual machines. Its primary responsibility is to virtualize the character device files

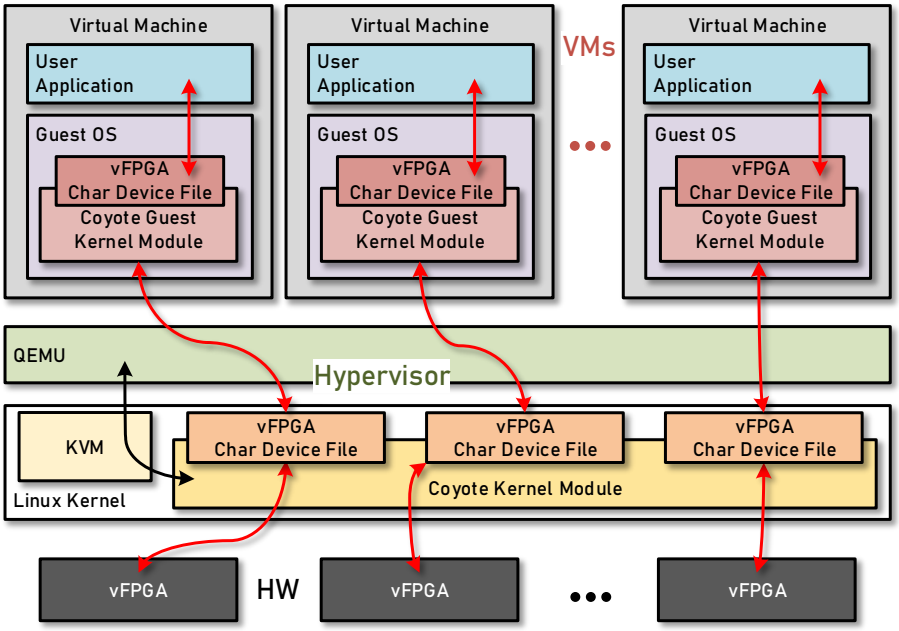

Figure 5.10: Virtual machines in Coyote.

in the Linux kernel that represent the virtual FPGAs running in hardware. The hypervisor is designed with the following set of guidelines:

1. It's crucial to maintain a consistent interface for these device files within the guest VMs. This ensures that users can run their code seamlessly, whether it's within a VM or in the native OS. To achieve this, we need to preserve the same set of standard system calls used by Coyote's API (predominantly based on *ioctl* and *mmap*).

2. Equally important is the preservation of isolation properties, both in hardware and software. Each VM is mapped to one vFPGA at a time to prevent interactions with other VMs or other vFPGAs.

3. Effective memory isolation is essential as well; users of one VM should not have access to the memory of other VMs. Coyote's custom MMU architecture, which handles protection on a per vFPGA basis, inherently supports all the necessary features for this functionality. Unlike Optimus, which required workarounds such as page-slicing to manipulate the host IOMMU, Coyote already supports memory isolation between multiple vFPGAs (the benefit of custom IOMMU).

4. Preserve performance; our primary goal is to accelerate tasks using FPGAs. If excessive kernel overhead due to numerous traps leads to a major performance loss, our virtualization efforts would be counterproductive. To avoid this, we must enable direct hardware control within VMs for the corresponding vFPGAs, removing kernel overhead.

5. Memory translation needs to be efficient. Mappings in the TLBs should mirror shadow page tables and translate directly from guest virtual addresses to physical addresses (eliminating the need for an extra translation step to host virtual addresses in between).

The virtualization process leverages the established combination of KVM for CPU and memory virtualization and QEMU for FPGA device and tenant virtualization. The primary role of the hypervisor is to ”lift” the virtual FPGA device files into the guest virtual machines. This is achieved by mimicking standard PCIe emulation within a guest VM.

The guest driver in the VM will identify this PCIe pass-through device and make it accessible through another character device file within the guest VM. Essentially, this PCIe emulation establishes a bridge between the VM and the hypervisor, enabling communication between character device files from both the guest and the host. This guest driver runs exclusively inside the VM and is implemented as a regular PCIe driver.

5.2.3.1 PCIe Emulation and Interrupt Forwarding

To enable PCIe device pass-through from the hypervisor to the guest VM, we utilize QEMU, which manages all accesses to emulated devices. Additionally, we leverage the existing API in the Linux kernel, VFIO [142], to allocate and share host PCIe resources among different VMs.

The guest driver in the VM detects and probes the emulated PCIe devices, which leads to the creation of a character device file. This character device file provides an interface to the user that resembles the one used in the non-virtualized Coyote driver. However, the driver in the VM operates in a virtualized environment and communicates with the hypervisor by triggering traps through memory accesses to designated physical addresses. This mechanism enables the driver to capture and handle device-specific operations, including registration and mapping procedures.

Accesses to these emulated PCIe devices are controlled by QEMU. The responsibility of the guest driver is to determine whether user accesses should be routed to the hypervisor for further mediation, or if they have permission to directly interact with the physical PCIe device, forwarding the accesses directly to the actual vFPGAs. This setup thus gives us two modes

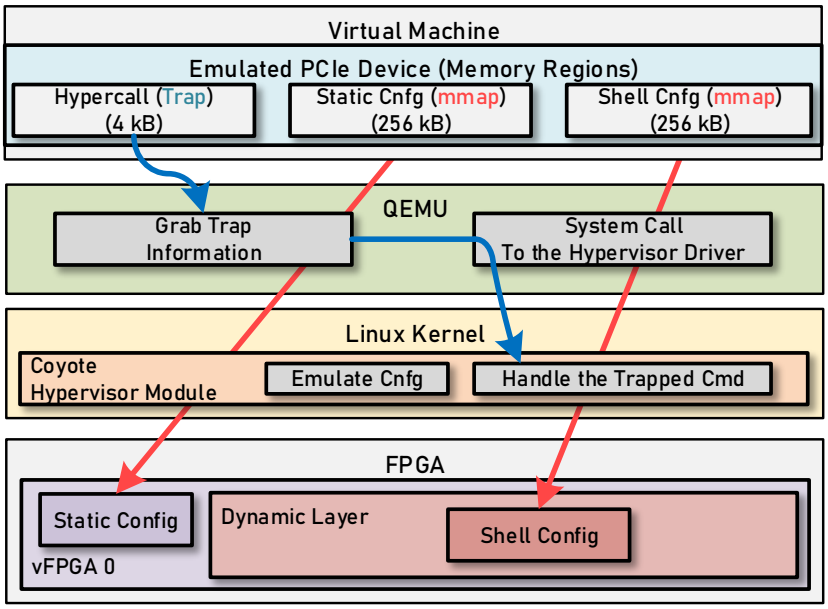

Figure 5.11: PCIe accesses: trapping and direct forwarding to hardware.

of operation: mediated accesses, which are trapped and handled by the hypervisor, and direct memory-mapped accesses that go straight to the vFPGAs with no overhead.

To allow using different vFPGAs in different VMs, we have to be able to emulate multiple PCI devices at the same time. This is done by exposing multiple device files, each implementing the VFIO API that can then be used by different QEMU hosts to enable the usage of different vFPGAs in different VMs. A device file that implements VFIO allows directly accessing device resources using reads and writes at specific offsets to the device file. In the case of PCIe devices, this means that we can access the internal memory regions of the device (PCIe Base Address Registers) directly from the user space.

We utilize different memory regions in an emulated PCIe device to detect which VM is making the access and where the access is destined. This allows us to mediate and route the accesses to either the corresponding vFPGAs directly or to the underlying hypervisor, while additionally providing all the information on the access that the hypervisor might need to resolve it. All of this is done completely transparent to the end user in the guest VM.

We also need to address the signaling in other direction, from vFPGAs to the VMs through interrupts. In this case we need to ensure that only the correct VM, the one owning the vFPGA, receives the interrupt. One possible approach is to signal the interrupt using eventfd, a

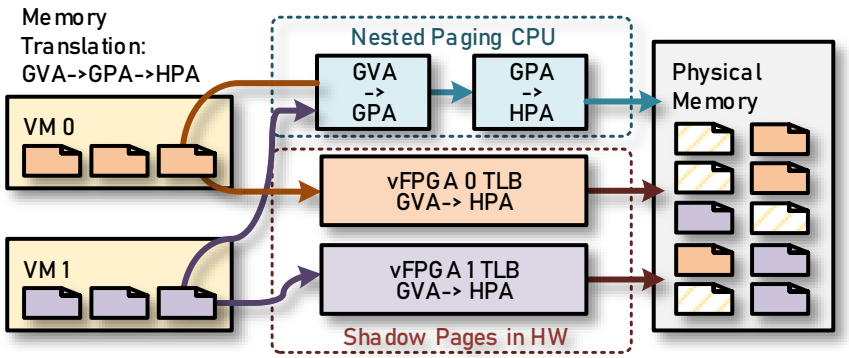

Figure 5.12: Memory translation in Coyote.

mechanism already employed in Coyote for notifying user space from vFPGAs (notification interface in ULI). In this method, we can signal QEMU, which runs in user space, and QEMU can, in turn, signal KVM to interrupt the VM. However, this approach introduces some additional overhead, as the context moves additionally through user space and then back into KVM. An alternative solution is to use the KVM irqfd library, which enables direct signaling of KVM from the hypervisor's interrupt handler. KVM will then forward the interrupt to the VM, and the VM can handle it similarly to the interrupt handler in the base Coyote driver. This approach incurs significantly less overhead due to fewer context switches.

5.2.3.2 Memory Mapping

It's also important to note the necessary changes in memory handling within the hypervisor. In the base Coyote driver, the mappings installed in TLBs translate from host-virtual addresses to host-physical addresses. However, in this context, these mappings must translate from guest-virtual to host-physical addresses. Therefore, when any explicit memory mapping by users occurs (or page faults), the guest driver must temporarily pin the pages inside the VM, which provides the necessary guest physical addresses. This also ensures that these pages are not swapped out during subsequent hypervisor page table walks. After obtaining guest physical addresses, the guest driver accesses the emulated PCIe device at a specific memory region, which triggers a hypercall (trap) into the hypervisor. This hypercall contains the necessary arguments for resolving the memory mapping, including the initial guest virtual and obtained guest physical addresses.

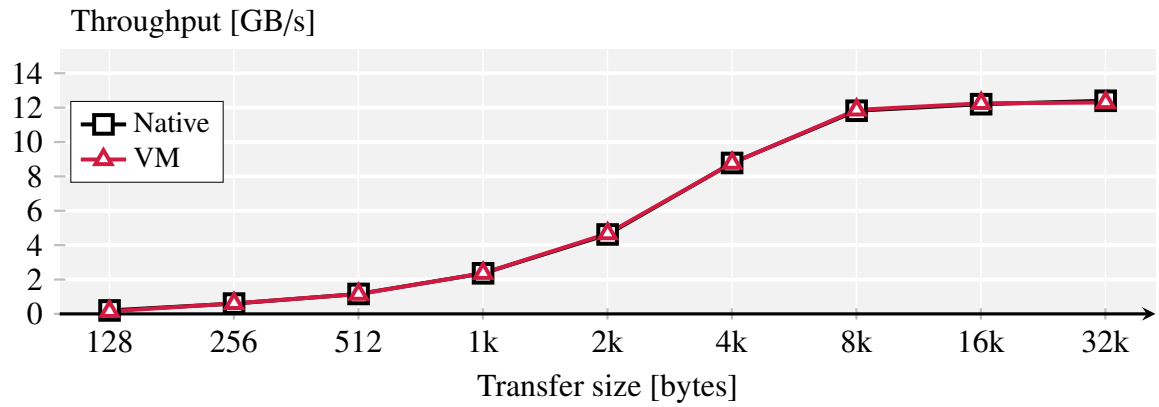

Figure 5.13: Host link throughput comparison: native vs VM (1 vFPGA).

Within the hypervisor, these guest physical addresses are then used to obtain host virtual addresses (using the gfa_to_hva function). With these host virtual addresses, another page table walk occurs, this time in the host OS, to obtain the physical addresses that need to be mapped into the TLBs. Finally, the mappings are executed in the TLBs through dedicated control lines, same as in base driver.

It's important to emphasize that despite the increased complexity of the host software driver code, there are no changes to the hardware. Coyote seamlessly supports this type of virtualization without requiring any modifications to the underlying hardware.

5.2.4 Evaluation

We will now run the benchmarks to estimate if virtualization adds overhead to normal operation in native system. All experiments are performed on Alveo u55c machines present at ETH cluster, which were already used in previous experiments.

5.2.4.1 Microbenchmark

In this first microbenchmark, we will assess the performance of the CPU-FPGA link in the virtualized environment. The objective of this test is to determine whether virtualization introduces any additional overhead to the overall performance. Our primary focus is to establish whether users (including virtual machines) can fully leverage the host CPU-FPGA interconnect for conventional acceleration operations. We execute a simple loopback test, where data is read from and simultaneously written back to the host memory. The comparison is made between

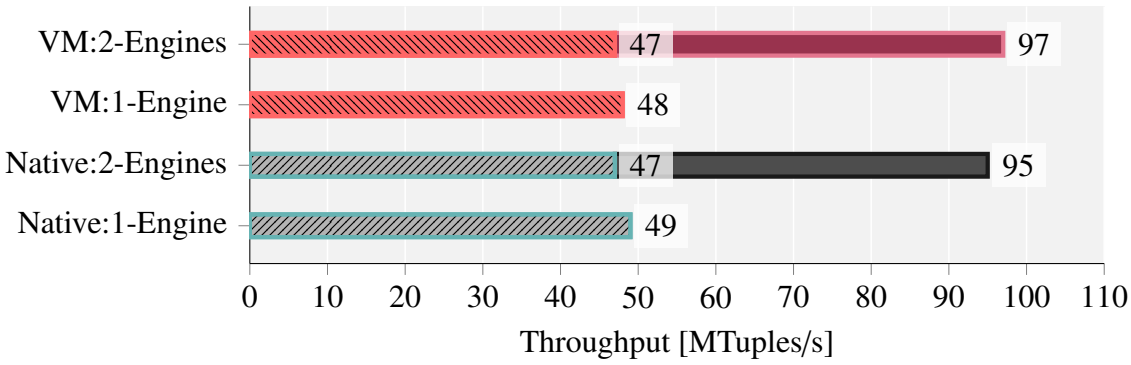

Figure 5.14: Decision trees: VM vs native (1 and 2 vFPGAs).

the native Coyote environment (no VMs) and a single virtual machine running on a KVM on the host (both cases take control of one vFPGA). The test's throughput results are presented in Figure 5.13.

The result demonstrates that when a single virtual machine runs on a single vFPGA, it achieves nearly identical throughput and latency to a scenario where no virtual machine is active. This outcome is of significant importance because it illustrates the effectiveness of the bypassing techniques we employ. These techniques involve directly memory mapping most of the control registers on the critical path into the user space of the virtual machine through QEMU. This approach eliminates all the overhead associated with the additional virtualization layer. As a result, KVM and the full kernel are effectively bypassed, allowing us to achieve near-native performance without incurring significant performance penalties. Simultaneously, we are still able to provide an extra layer of virtualization and isolation to end-user applications.

5.2.4.2 Concurrent VMs

We will now examine whether this outcome remains consistent when multiple virtual machines are involved. In this instance, we will conduct the same test but with two virtual machines and two vFPGAs. For this test, we will use the same Gradient Boosting Decision Trees that we used for the evaluation of Coyote's memory model. It's worth noting that this test employs the same bitstream generation that was done for the experiments on Enzian in Section 4.2.8.2, underscoring that no modifications to hardware code were required to accommodate the additional virtualization layer.

We conduct a comparison of throughput (per vFPGA) between native execution (utilizing one and two engines) and running with one or two virtual machines. The results are shown in

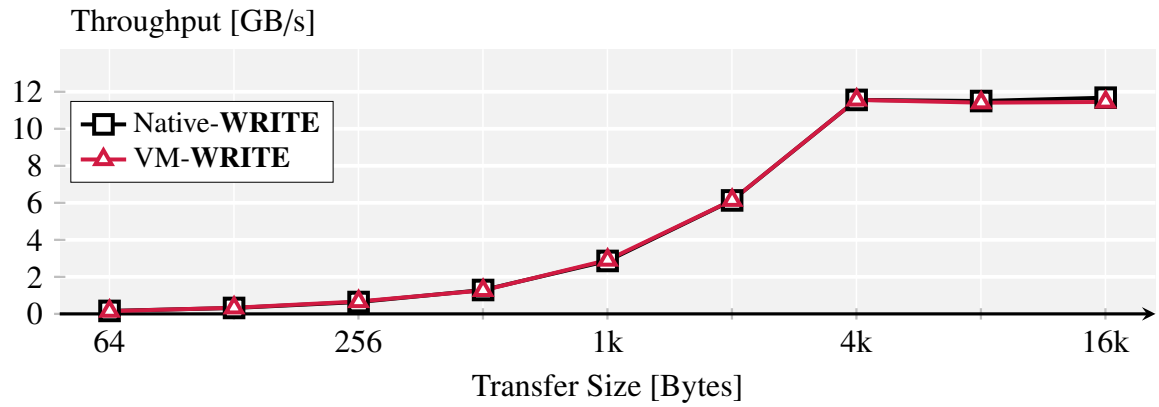

Figure 5.15: RDMA Throughput comparison: VM vs native (1vFPGA).

Figure 5.14. Much like the microbenchmarks, we once again observe minimal differences in performance. Furthermore, we notice that we achieve nearly perfect linear scaling when using two virtual machines, mirroring the behavior observed in native execution. This underscores that, even for real-world applications, we experience almost negligible virtualization overhead.

5.2.4.3 RDMA Performance

Lastly, we aim to demonstrate that the additional software virtualization layer not only offers a high level of abstraction and isolation for a single machine but extends its benefits to scenarios involving multiple machines connected over a high-speed networking fabric. In this scenario, we have two machines linked by high-speed networking and intend to facilitate data exchange between users, each of whom has been allocated their own respective virtual machines on these different physical machines. These users wish to exchange data at high speeds, and for this high-performance communication, they opt to utilize RDMA (Remote Direct Memory Access).

Given that RDMA operates at the user space level, it bypasses not only the lower layers of the host kernel stack in Coyote but this time also includes the software virtualization layers added to support the virtual machines. Figure 5.15 presents the throughput comparison between the network communication (RDMA WRITE) executed on native Coyote and the same communication between two users while they are running inside their respective virtual machines. Crucially, as observed in our other experiments, we detect virtually no overhead. This demonstrates our capability to achieve a high degree of isolation provided by both virtual machines and Coyote's framework, while still being able to leverage the full networking performance.

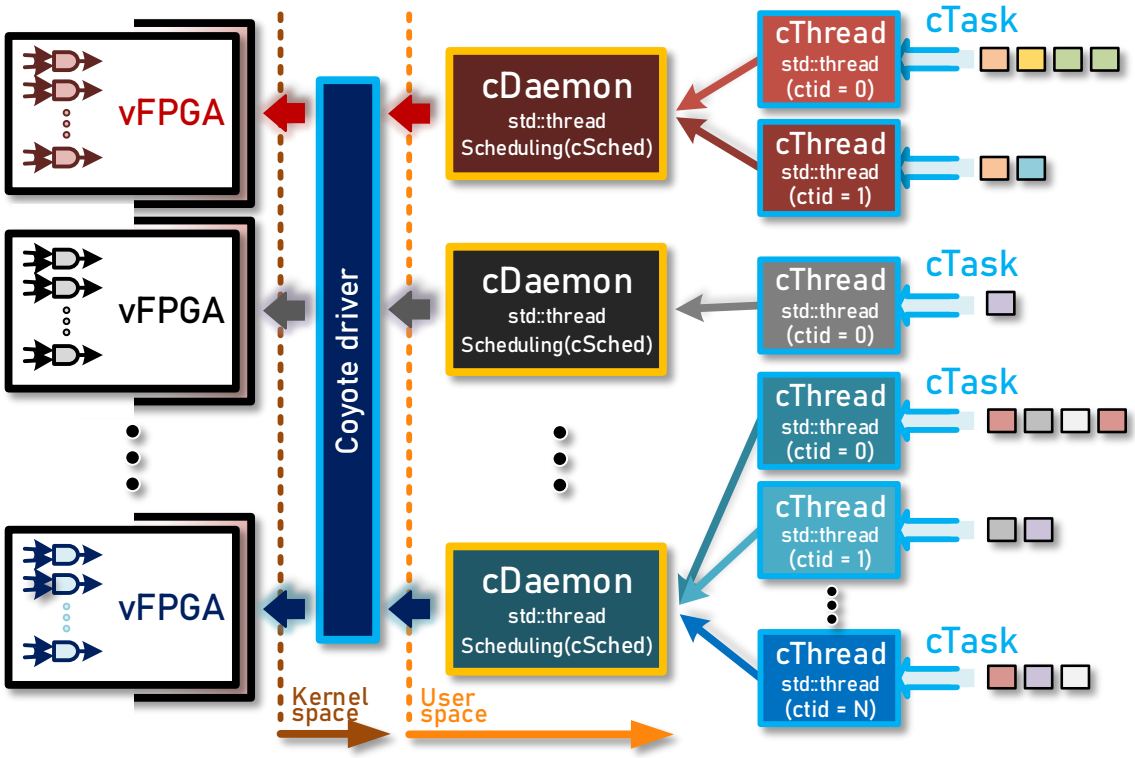

Figure 5.16: Software API architecture.

5.3 Software API

We will now explore the API accessible to end users who are writing software targeting the heterogeneous hardware. At this stage, we have addressed not only the hardware abstractions but also the virtualization layers within the software, which collectively comprise the interface for developers and users creating conventional code that interracts with the framework.

Coyote's API is predominantly created using the C++ programming language. The main factors behind opting for C++ include its cross-platform compatibility, focus on high-performance applications, support for low-level operations (particularly fine-grained memory control), all of which are advantageous for Coyote. Furthermore, C++ offers a range of higher-level abstractions compared to standard C, which contribute to a more streamlined development process. We also conducted some preliminary experiments with possible bindings for other programming languages (Go for instance). We plan to explore additional programming languages, with a par-

ticular emphasis on porting the API to Rust. Rust is of significant interest as it appears to be a modern and fitting replacement for C++ [57], especially for a system like Coyote.

When examining traditional interaction with FPGA hardware, it typically involves low-level, bare-metal interfaces where the primary assumption is that the FPGA serves as a single "dumb" accelerator attached as a slave device to the host CPU. In this setup, the CPU offloads data for acceleration, awaits completion, and then retrieves the results.

However, this deployment model often overlooks the fact that, in addition to a powerful FPGA, there is typically a highly capable CPU (e.g., Enzian's 48-core CPU). Therefore, beyond maximizing parallelization and multitasking in the FPGA (a primary focus of Coyote), it is essential to consider how to provide abstractions that parallelize execution on the host CPU simultaneously. This was the core concept behind the development of the Coyote API: to offer complementary set of multitasking abstractions that are distributed across both the CPU and the FPGA.

Our exploration of the API will start by focusing on the foundational element in Coyote's API: the Coyote thread.

5.3.1 Coyote Threads and Tasks

As previously established, Coyote's hardware primarily revolves around a single virtual FPGA, which serves as the fundamental abstraction. These vFPGAs play a vital role in ensuring isolation within both the FPGA hardware and the memory hierarchy of the system. Given this context, it might be tempting to directly assign each of them to individual host processes. However, in a practical sense, we are limited to deploying only a small number of these vFPGAs at once, typically no more than 10. This limitation can be rather restrictive and prevents us from fully exploiting the parallelization potential that FPGAs offer.

The additional potential for parallelization primarily resides in the FPGA hardware's capacity for comprehensive pipelining. This enables multiple threads of execution to advance through the pipelines stages without needing to wait for the completion of their predecessors. This is a crucial characteristic for achieving a higher level of parallelization in the FPGAs. This form of pipelined parallelism is a key advantage that FPGAs possess over accelerators like GPUs.

This is where Coyote threads, which we have already mentioned quite a few times in this thesis, enter the picture. They enable us to "move" multiple threads of execution through the shared pipelines within the same vFPGA (instead of being limited to a single thread of execution).

This is achieved while maintaining the essential distinction between the individual threads for all the data words entering these pipelines, allowing for straightforward differentiation and data manipulation. A similar concept (to an extent) is already well-established in modern superscalar CPUs, where hyperthreading allows multiple threads to run on each CPU core simultaneously to reduce data access latencies. One can argue that this idea is inherent to FPGA accelerators, especially since they primarily extract performance from deep processing pipelines.

5.3.1.1 Implementation of Coyote threads

Every Coyote thread is symbolized by a dedicated software object (*cThread*). This object serves as the fundamental building block of the software API, providing users with the means to interact with vFPGAs in hardware.

When a thread is initially created, it is associated with a specific vFPGA. The thread's constructor will first open the `vfpga_dev` character device file in the kernel, immediately followed by an explicit system call for registration of the new Coyote thread. This registration process allocates a unique ID (*ctid*) to the thread, which is then returned to the user space. At the same time, during the registration, host process ID is provided which will be registered to this thread in the Coyote driver. This host process corresponds to the actual user application utilizing the vFPGA. The memory pages of this process will be accessible to Coyote thread (note that this process may not necessarily be the one that initiated the system call, as we'll see in the Section 5.3.2).

During this initial call to the constructor, users can also provide a pointer to the scheduler object responsible for handling any required dynamic reconfigurations. This scheduler implements the modified priority-based queue scheme discussed in a previous chapter. In addition, users have the option to supply a pointer to a function that will execute asynchronously when a vFPGA notification (interrupt) is received. The implementation of this function can be tailored to suit the user's specific needs.

Following the registration process, the thread will establish a series of default direct memory mappings (through *mmap* system calls). These mappings enable the thread to access all the necessary resources required for direct interaction with the vFPGA. It's essential to emphasize that these Coyote threads can exist in different host processes, and multiple threads can also be instantiated within the same process. Therefore, they are not tightly bound to any specific user space OS abstraction.

For synchronization between these Coyote threads running within a single vFPGA (whether they are within the same process or across different processes), named mutexes are employed in user

space. These mutexes are implemented in shared memory and are managed by the API (not directly exposed to end users). Since they reside in shared memory, they do not incur any kernel overhead for managing inter-process communication.

Listing 5.3: Coyote thread API.

```
class cThread {
...
private:
    void processRequests(); // Process submitted user tasks

    std::thread notify_thread; // Thread for notifications from vFPGAs
    std::thread cyt_thread; // Coyote thread (run user submitted tasks)
...
public:
    // Constructor, pid can be from different process
    // The scheduler is responsible for dynamic reconfiguration of user applications
    // Uisr is a function pntr to custom interrupt routine when notified by vFPGA
    cThread(int32_t vfid, pid_t pid,
        cSched *csched = nullptr, void (*uisr)(int) = nullptr);

    // Control/Status registers
    inline auto setCSR(uint64_t val, uint32_t offs) { ctrl_reg[offs] = val; };
    inline auto getCSR(uint32_t offs) {  return ctrl_reg[offs]; }

    // Invoke all data operations in Coyote (local or remote data transfers)
    // csInvoke is a struct describing the operation
    void invoke(const csInvoke& cs_invoke);
    // Check for completion of the operations
    uint32_t checkCompleted(CoyoteOper coper);

    // Start the thread
    void start_thread();

    // Submit user tasks
    void scheduleTask(std::unique_ptr<bTask> ctask);
}
```

After the initial object construction, a small set of functions becomes available to users for direct interaction with the vFPGA. The primary functions in the API are shown in Listing 5.3. One of the key functions is the *invoke* function, which serves as the primary means for initiating all data transfers within Coyote, both locally and remotely. The details of the operation to be executed are specified through the *csInvoke* struct. Importantly, this struct also includes all the virtual addresses necessary to identify the buffers for the transfers, whether they are local or remote. This highlights the benefit of the shared virtual memory model, as users are freed from the need for additional explicit memory allocations or unnecessary memory copies; they can simply utilize virtual pointers directly.

In addition to the *invoke* function (and its corresponding completion function), there is a function which directly accesses control/status registers in the vFPGAs. Users can employ this to explicitly control the operation of their hardware applications. There are some additional functions in the API besides these, such as explicit memory mapping or network connection establishment operations, but they hold less importance in this context. Thus, compared to the typical FPGA interaction, Coyote's abstractions rely on far fewer functions in the API, making the interaction much simpler.

Listing 5.4: Coyote task template.

```
template<typename Func, typename... Args>
class cTask : public bTask {
private:
    std::tuple<Args...> args;
    Func f;
public:
    explicit cTask(int32_t tid, int32_t oid, uint32_t priority, Func f, Args... args)
        : f(f), args{args...}, bTask(tid, oid, priority) {}

    virtual int32_t run(cThread* cthread) final {
        int32_t tmp = apply(f, std::tuple_cat(std::make_tuple(cthread), args));
        return tmp;
    }
};
```

5.3.1.2 Coyote Tasks

However, the set of functions shown in Listing 5.3 is not the final level of abstraction in software. As the name implies, these Coyote threads are not just threads in the hardware context, but are also just regular software threads as well. These objects encapsulate standard POSIX threads [1] and are responsible for executing user-submitted tasks via the *submitTask* function. These tasks are essentially wrappers around user-defined lambda functions, which include a pointer to a Coyote thread as their first argument. These threads execute the lambdas while setting the first argument (the handle to the Coyote thread) to themselves. Crucially, before executing these tasks, a Coyote thread contacts the scheduler to request the loading of a specific hardware operator (partial bitstream) required for the task. The task is executed only after receiving approval from the scheduler (bitstream has been loaded). As a result, Coyote threads, in conjunction with the scheduler, orchestrate the execution of these user-submitted tasks.

This approach enables writing code (tasks) in a manner that separates the code from the actual execution. Users don't have to concern themselves with the exact thread in which their task

will be executed or manage the complex details of dynamic reconfiguration, scheduling and ultimate loading of target bitstreams. This can simplify the management and distribution of these tasks throughout the system, making it easy to build things like load balancers for instance, without pushing the complexity of these implementations to the end users. An example of a task performing K-Means clustering on a dataset, is provided in the Listing 5.5. In this particular task, the impact of unified memory is also clearly demonstrated, as users can direct data where needed (*CARD* and *HOST* flags) without having to allocate and manage explicit buffers and data migration between different devices.

Listing 5.5: K-Means task.

```
template<typename Func, typename... Args>
cthread->submitTask(opKMeans, tid, priority,
[] (cThread *cthread, kMnsStruct prms, void *cntMem, void *dataMem, void *rsltMem)
-> int32_t {
    // Set parameters
    cthread->setCSR(prms.data_set_size, KMEANS_DATA_SET_SIZE_REG_OFFS);
    cthread->setCSR(prms.n_iterations, KMEANS_N_ITER_REG_OFFS);
    cthread->setCSR(prms.n_clusters, KMEANS_N_CLUSTERS_REG_OFFS);
    cthread->setCSR(0x1, KMEANS_CTRL);

    // Stream initial clusters
    cthread->invoke(
        {CytOper::LOCAL_CARD_READ, centMem, prms.n_clusters * sizeof(uint64_t), false}
    );
    // Stream data
    for(int i = 0; i < prms.n_iterations; i++)
        cthread->invoke(
            {CytOper::LOCAL_CARD_READ, dataMem, prms.data_set_size * sizeof(uint64_t)}
        );
    // Write results (blocking, poll for completion)
    cthread->invoke(
        {CytOper::LOCAL_HOST_WRITE, rsltMem, prms.n_clusters * sizeof(uint64_t), true}
    );

    return 0;
})
```

5.3.2 Coyote Daemon

The tasks offer an intuitive way to work with the hardware and abstract away much of the complex management happening within the framework. However, these tasks still require their creators to understand the hardware's operation and the interface it exposes (as evident from the

K-Means example code). This level of abstraction might not be suitable for end users who just want to run a specific acceleration function without being concerned about the details of how it is executed.

For these end users, a further layer of abstraction is provided through the Coyote daemon object. This object launches a system-wide Coyote service in the background, which listens for periodic service requests from applications running on the same system. It can be viewed as a system-wide library of accelerators that is offered as a service to end users. These end users don't need to be well-versed in how vFPGAs and their hardware components function. Their only job is to invoke tasks with specific arguments and specifying the required hardware operator needed for the execution (target bitstream).

The Coyote daemon is initially loaded with a range of bitstreams (hardware operators) that can be loaded on demand onto the available vFPGAs. The Coyote daemon then manages Coyote threads for each end user process that registers with the service. Communication between end user applications and the Coyote daemon is facilitated through standard Unix Domain Sockets, which handle all inter-process communication in this case.

Since the Coyote daemon acts as a middleware, scheduling tasks from end user applications to Coyote hardware, the end user applications themselves don't require special security permissions to execute their requests. Ultimately, it's the Coyote daemon that interacts with the Coyote driver and issues system calls. This introduces an extra security layer that separates end users from the lower parts of the stack, and can also be employed to incorporate various additional features or functionalities.

This also explains why, as mentioned in the previous section, we can register Coyote threads for processes that are not the ones directly making the system calls. In this case, the system call is made by the Coyote daemon, while the process ID (pid) and memory regions to be accessed, belong to the end user application that requested access.

Listing 5.6: Request to Coyote daemon to execute K-Means.

```
1 cLib clib("/tmp/coyote-daemon-vfid-0"); // Registration
2 clib.task({opKMeans, {params, cntMem, dataMem, rsltMem}}); // UDS request, blocking
```

Returning to the example of K-Means mentioned earlier, in this scenario, an end user who wants to execute the same operator only needs to write the lines of code shown in Listing 5.6. This is the only interaction an end user has with the system, so there's no need for an extensive understanding of the system, nor does it involve much complexity.

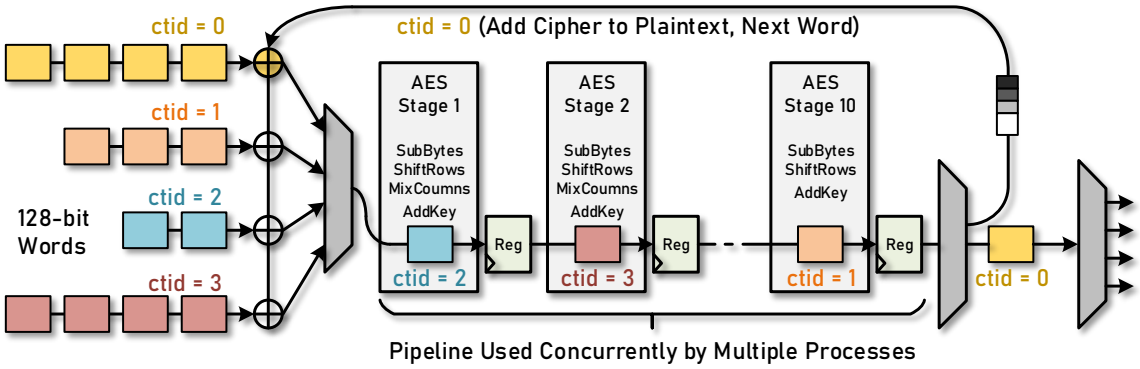

Figure 5.17: AES "multithreading" example.

5.3.3 Multithreading

Now that we've discussed the software API and Coyote threads, let's return to the hardware aspect and explore how concurrent Coyote threads can be effectively used in certain scenarios to better leverage the available performance in the hardware.

We'll revisit our AES encryption core, which we've used in various tests throughout this thesis. This time, we'll focus on the sequential CBC mode, which isn't parallelizable. Our AES core takes 128-bit data input and consists of 10 sequential pipelined stages of computation. Each 128-bit data word within a single thread of execution has a dependency on the previous 128-bit data word. This means that each subsequent 128 bits must wait for the preceding 128 bits to complete the full pipeline. If there are other threads of execution waiting for the same computation, they will be stalled until the entire computation is finished. Consequently, a substantial portion of our pipeline will remain empty and underutilized (9 out of 10 stages).

To address this issue, we can pass multiple threads of execution (from different processes) through the pipeline while keeping track of which data belongs to which thread at any given point. Each data word contains an association with a specific thread ID (*ctid*), which makes it easy to differentiate between data words, monitor their progress within the pipeline, and ultimately write them to appropriate locations.

In the simple example of this, we run the AES in CBC mode with 4 different user processes using the core, and submitting 4 MB encryption tasks continuously. First we will process these tasks fully sequentially, as would be the case in shells which support only a single thread of execution.

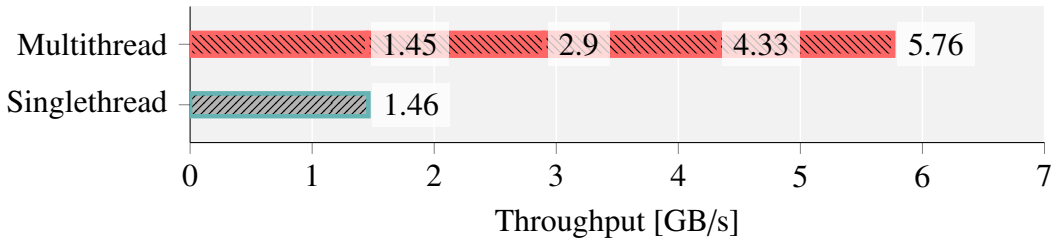

Figure 5.18: Single thread of execution vs multithreaded execution (AES CBC pipeline).

In the second case we will modify the core and utilize the abstractions in Coyote (Figure 5.17). We will use 4 different Coyote threads in these 4 different processes to interract with the vFPGA concurrently. Then we will use the destination queues to interleave the requests. In this way data will be fetched from the memory and thus available in different queues even if the previous requests were not fully completed.

The achieved throughputs of the two tests are shown in Figure 5.18. The higher throughput of the accelerator in the "multithreaded" case is a direct result of better pipeline utilization. It's important to note that this is a simple illustration, and while similar pipelined parallelism might be feasible to exploit in other systems, it's not common (to the best of our knowledge) to find systems that support multiple concurrent host processes accessing the same hardware operator. The advantage here lies in the available abstractions, which simplify the design of such circuits and the management of these operations. With these abstractions, there is no need to worry about how to manage and identify multiple data sources, protect memory accesses of different processes or handle request interleaving, making the process much more straightforward.

5.4 Use Case - ACCL

The question that emerges is: what practical applications can now effectively leverage these abstractions that Coyote offers? A great illustration of a higher-level framework that can make use of Coyote is ACCL [105]. ACCL serves as an FPGA-based collective offload engine for distributed applications. We port this framework to the Coyote infrastructure, thus allowing it to fully exploit the flexibility of modern FPGA devices. We also conduct a direct comparison to assess the benefits that Coyote brings to ACCL, as opposed to its previous operation within a state-of-the-art vendor shell.

ACCL extends the networking abstractions beyond the existing networking services in Coyote. In many scenarios, the networking services available in Coyote, such as TCP and RDMA, lack

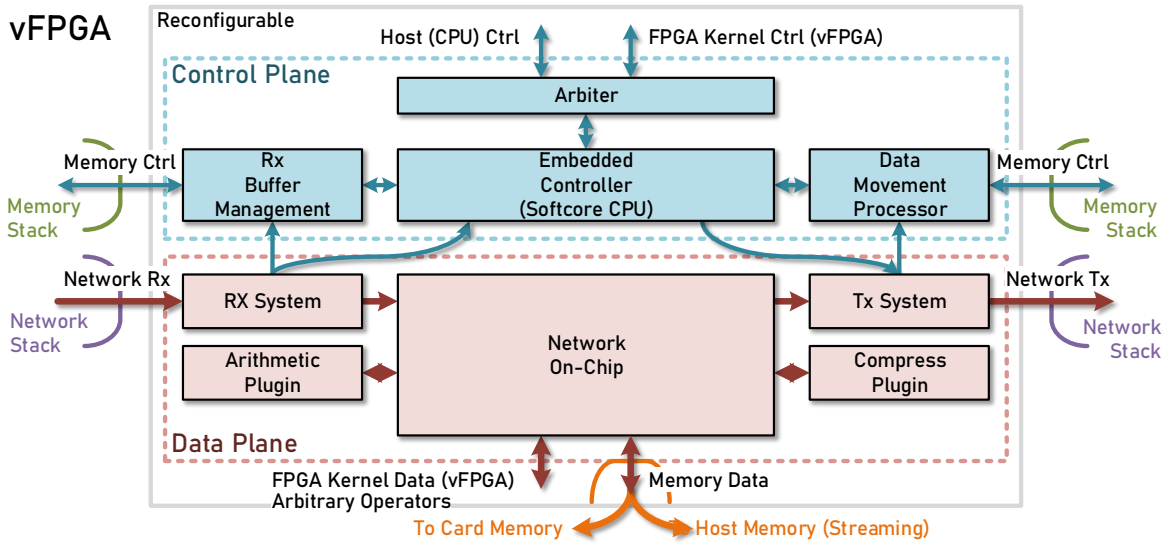

Figure 5.19: ACCL internal architecture.

the higher-level abstractions required for designing distributed applications. In the context of general-purpose hardware, high-level collective communication and data synchronization operations, such as those provided by MPI (Message Passing Interface), are commonly utilized and widely available within the system libraries. These operations can greatly simplify the process of developing distributed applications, especially when handling complex data exchange patterns (scatter, reduce, etc.) among a substantial number of distributed machines. These kind of abstractions are missing in modern FPGAs and are the main contribution of ACCL.

ACCL was initially designed for use with the AMD XRT shell [208] and operates within the Vitis [207] environment. It's important to emphasize that this environment is not the standard Vitis shell but rather an enhanced shell [104] that provides the essential TCP support that the base shell lacks. This extended shell utilizes the same base TCP network stack as Coyote.

The architecture of the ACCL hardware operator is shown in Figure 5.19. ACCL provides MPI-like functionality for memory-mapped applications and is characterized by a highly modular design featuring multiple abstraction layers. It is a "true" heterogeneous framework, serving both software applications running on the host CPU and hardware applications executed on the FPGA. As a result, it is a highly compatible system for operation within the Coyote environment. ACCL exposes a straight forward software API that offers collective operations commonly found in MPI libraries, such as reduce, broadcast, and gather to end user applications.

The primary control within ACCL's hardware is managed by the softcore CPU in the FPGA

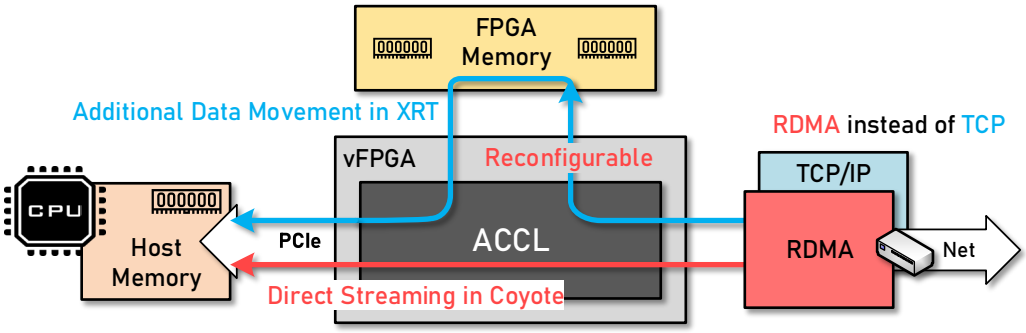

Figure 5.20: Abstractions in use by ACCL in Coyote.

fabric, which significantly reduces overall resource consumption and, consequently, the cost associated with ACCL. This FPGA-side CPU is responsible for managing the entire operation of the collectives, from their initiation to completion. Its primary role involves control of the overall data movement, primarily through dedicated DMA operations, spread out across the hardware platform. Additionally, the MPI-like collectives provided by ACCL can be triggered from either the host CPU or directly from the FPGA fabric, further improving the flexibility.

Another significant advantage of ACCL, when compared to traditional MPI implementations over NICs, is the ability to seamlessly integrate additional processing directly into the path of these collectives within the FPGA. This enables users to conveniently extend the existing collectives and conduct additional pipelined processing as required.

In the context of this thesis, we won't dive into the finer architectural details of ACCL, as our primary focus is to demonstrate how ACCL can leverage a system like Coyote to significantly enhance its flexibility and, ultimately, its performance.

5.4.1 Abstractions Used

What abstractions within Coyote are beneficial for ACCL?

Direct Streaming

The first interface utilized by ACCL to increase performance, as opposed to its operation in the XRT shell, is the streaming interface provided by Coyote. This interface facilitates direct data transmission for collectives to the host memory, eliminating the need for an intermediate step of storing data in the local FPGA-side memory. For instance, in a scenario where data is received from a remote node, this approach bypasses the extra copying step and directly transfers data

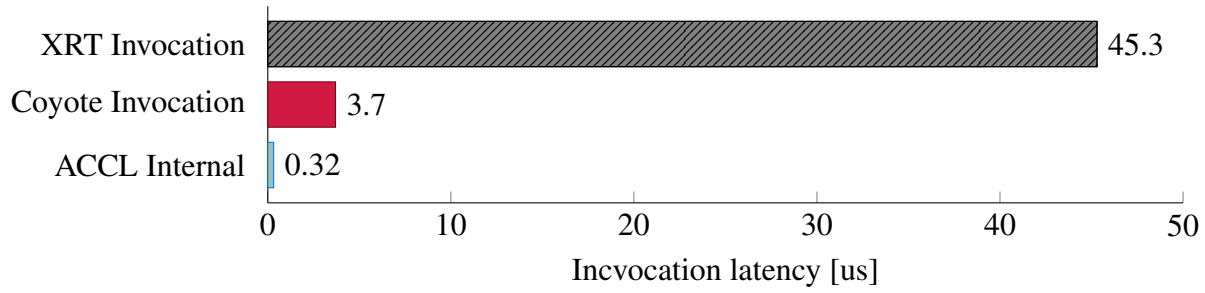

Figure 5.21: Kernel invocation times, comparison between Coyote and XRT.

to the host memory without any additional interruptions. Simultaneously, it's possible to enable multiple parallel data streams directed to the FPGA memory, offering ACCL added flexibility in determining how to partition and employ the buffers between host and FPGA memories.

This interface significantly reduces overhead, not only by eliminating extra data movement to the FPGA-side memory but also by avoiding the subsequent data migration to the host (or from the host), which, in the case of XRT, requires major additional CPU involvement, thus negating the advantages of collective offload in the first place.

Lower Hardware Invocation Time

The second significant factor influencing performance is the total time taken for the host CPU to invoke the hardware operators. This invocation encompasses the time required to issue a series of sequential commands that are aimed at configuring and initiating the operations within the hardware operators. This time is primarily constrained by the speed of the host CPU-FPGA interconnect and, in the case of PCIe, typically falls within the microsecond range.

The benefit of our approach in Coyote is that these commands are issued through a memory-mapped bus in the user space, which eliminates any potential kernel overheads. Consequently, the speed of issuing individual commands approaches the lower bound of PCIe. In contrast, with the XRT shell, due to the absence of kernel bypass, every command first needs to undergo a context switch into the kernel space, which introduces a significant amount of overhead. The comparison of the time it takes to set up and start the operation of a collective in ACCL between Coyote and XRT is illustrated in Figure 5.21.

RDMA

Another advantage of Coyote lies in its RDMA stack and the high-level networking abstractions it offers. For a system like ACCL, which is built around networking communication using memory-mapped buffers, this abstraction is crucial for achieving optimal performance. This is

because it removes the need for extra control mechanisms within the FPGA to explicitly manage networking data.

The XRT shell exclusively supports TCP and does not offer RDMA support. The reason for this limitation is that XRT does not expose the host memory hierarchy and lacks any virtualization layers on top of FPGA-side memory. As a result, interaction with memory in XRT from hardware applications is limited to the physical memory on the FPGA side.

In contrast, RDMA demands additional levels of virtualization, a dedicated IOMMU, and the ability for the FPGA shell to independently handle transactions (without CPU involvement). None of these components are available in the XRT shell due to its constrained memory model.

Therefore, TCP remains the sole option, as it can handle the networking data exchange to and from the hardware operators while leaving any further interaction with the FPGA memory or host-side applications to be managed by the hardware operators themselves.

This ultimately implies that, in Coyote with the added RDMA support, there's no requirement for additional commands to be issued by ACCL to manage incoming or outgoing network packets. This information is inherently present within the packets, leading to a substantial improvement in overall system performance.

Dynamic Reconfiguration

Lastly, another beneficial feature is the application level dynamic reconfiguration. This allows us to easily replace the processing kernels running on top of ACCL collectives on-the-fly while keeping the remainder of the system operational. This implies that we can have several independent ACCL instances within a single FPGA. This flexibility enables a wide range of computations to be provided on-demand on top of these collectives. In contrast, this level of flexibility is not achievable in XRT, as it lacks fine-grained dynamic reconfiguration capabilities.

5.4.2 Resource Usage

The resource utilization of ACCL in Coyote is presented in Table 5.2. ACCL leverages various abstractions within Coyote, including parallel host and card memory interfaces with multiple data streams, all while utilizing RDMA for reliable communication. This serves as an excellent example of how extensively we can utilize the system when all these features are active.

ACCL, on its own, consumes less than 20% of the total resources. This low usage is primarily due to the offloading of the control plane to the softcore CPU in the FPGA fabric, which has minimal resource requirements. Approximately 40% is allocated to the shell and the static layer.

This leaves roughly 40% of resources available for arbitrary user-defined processing that can be integrated on top of the collectives.

This resource allocation highlights the fact that these abstractions are not without cost, and a significant portion of the FPGA must be allocated for the shell itself. However, this trade-off is deemed acceptable as it is essential to enable the desired end functionality on these devices.

At the same time, we notice that similar resource utilization persists in the base XRT shell as well. Therefore, Coyote offers similar resource usage as the XRT shell while providing a far higher level of functionality and features to the application.

Table 5.2: ACCL resource usage (Alveo-u55c).

Platform	CLBs	LUTs	Registers	Block RAM
Coyote-RDMA	96 667 (59.32 %)	416 395 (31.94 %)	759 263 (29.12 %)	443 (21.99 %)
Coyote-TCP	88 356 (54.22 %)	427 476 (32.79 %)	707 116 (27.12 %)	455 (22.59 %)
XRT-TCP	96 782 (59.4 %)	423 435 (32.48 %)	690 168 (26.47 %)	492 (24.43 %)

5.4.3 Evaluation

The evaluation of ACCL on Coyote is performed on a heterogeneous cluster, which has been the primary testbed used for most of the evaluations in this thesis. All devices in the cluster are connected through high-speed 100 Gbps links, with the CPUs linked to Mellanox NICs for this purpose. All experimental results are based on the average of 250 runs.

ACCL is evaluated in two distinct scenarios: one where data is stored in FPGA-side memory and another where it resides in host memory. Coyote inherently provides direct access to both of these memories, making this option transparent to ACCL. In Coyote, data is directly transferred from the network, passing through ACCL and reaching the intended target memory (for all receive operations, while the opposite direction is taken for sends).

In the baseline scenario involving host memory access, full data movement is managed by the Mellanox NIC, which manages the reading and writing of data to host memory. However, in the baseline cases where FPGA-side memory is utilized, the additional data transfer to/from the FPGA memory is coordinated by the CPU.

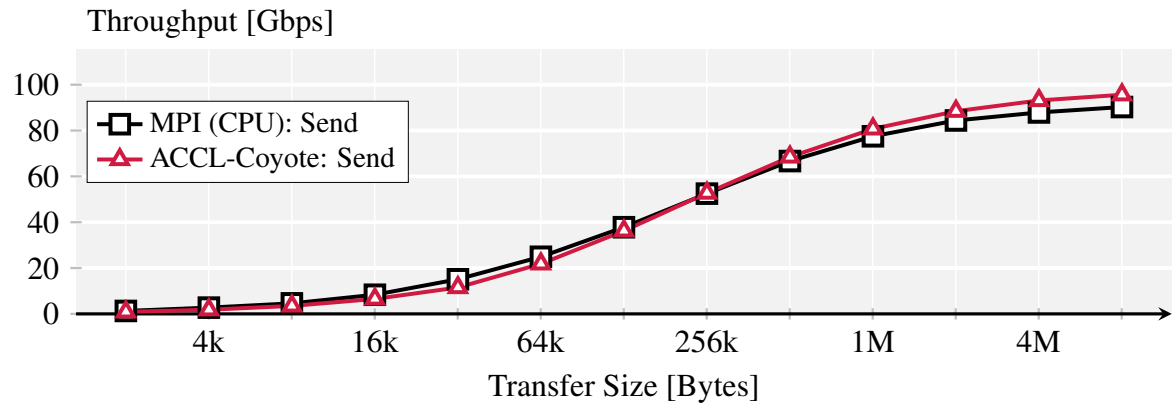

Figure 5.22: Send collective.

5.4.3.1 Microbenchmarks

To start, we will perform microbenchmarks aimed at evaluating the maximum throughput of essential data movement operations in ACCL, specifically the send and receive operations. For the baseline comparison, we will use a commercial Mellanox card executing equivalent operations using OpenMPI 4.1.3 [100]. The results of this test are shown in Figure 5.22.

As evident from the results, they closely resemble the ones obtained for the base RDMA performance in Coyote. This confirms the core concept behind the incorporation of the RDMA stack into Coyote. It enables the integration of various user applications on top of the RDMA, with minimal overheads. Similar to the base RDMA test, we also observe comparable performance to the commercial NIC, with ACCL operating on Coyote achieving a slightly higher peak throughput (as previously noted), thanks to the efficient pipelined execution in FPGAs. This supports the fact that FPGAs are not a bottleneck in modern network workloads.

5.4.3.2 Collective Performance

We will now briefly evaluate the performance of one of the frequently used collectives in ACCL, the gather operation. For this test, we have deployed 8 Alveo-u55c boards that are communicating with each other. Data is gathered from all other devices to one designated device. This test also shows how Coyote supports operation across a large number of devices in a cluster.

The performance of the collective operation for both host memory and FPGA-side memory is presented in Figure 5.23(a) and Figure 5.23(b), respectively. The results clearly demonstrate that ACCL on Coyote delivers substantial performance advantages when compared to software

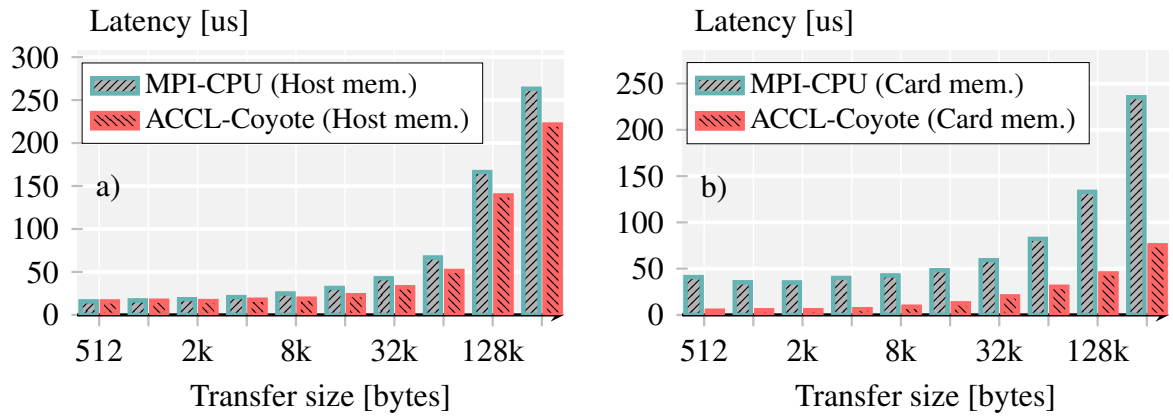

Figure 5.23: Gather collective with a) host memory and b) FPGA-side memory.

MPI libraries running on commercial RDMA NICs. The performance is not only superior when FPGA memory is used (where the baseline needs additional copying), but it also remains ahead when host memory is employed, showing that networking performance on FPGAs can compete with commercial NICs.

Large portions of these performance benefits, besides the efficient ACCL implementation, come from the fact that FPGAs are well suited for these kind of networking operations. Additionally, in Coyote we can transfer data to the desired locations without the need for extra copies and with minimal overheads, further improving overall performance.

5.4.3.3 Comparison to XRT Shell

Finally, we will perform a quick comparison between ACCL running on Coyote and ACCL running on the XRT shell. Since XRT does not support RDMA, the only fair option for comparison is to utilize TCP in Coyote as well (which limits ACCL performance in Coyote).

The results for the same gather collective, in the scenario where data is located in host memory, are shown in Figure 5.24. As evident from the results, ACCL achieves significantly higher performance with Coyote. This is primarily due to the considerably high kernel invocation overheads in the XRT shell and the substantial cost associated with migrations in the XRT due to the absence of direct host streaming interfaces.

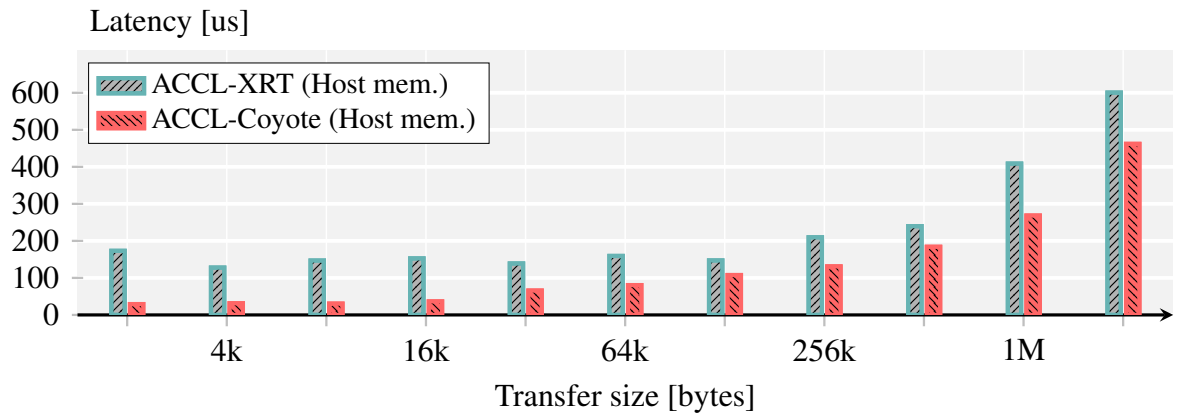

Figure 5.24: ACCL comparison: Coyote vs XRT (both TCP/IP).

5.5 Summary

In this chapter, we explored the user interface exposed by Coyote. We began by presenting an overview of the Unified Logic Interface, which serves as the standard execution environment and plays a pivotal role in ensuring high portability.

Next, we looked into the user wrapper, which encapsulates user-deployed hardware applications. We illustrated how these user applications are safely decoupled from the rest of the system.

The following section then assessed how the virtual FPGAs in Coyote can be leveraged within the software context of virtual machines, providing an interesting model for modern cloud deployments of FPGAs.

We proceeded to cover the software API and the final layer of abstractions within Coyote. Lastly, we showed a real-world application that maximizes the potential of Coyote, delivering substantial performance benefits.

Chapter 6

Farview: Smart Disaggregated Memory

While we have already explored how applications and additional abstractions can be executed and applied on Coyote, this chapter will take a broader perspective. We will investigate how the existing abstractions in FPGAs can be leveraged to build innovative systems for modern database workloads, with a particular focus on a promising research direction that is highly relevant in modern data center and cloud environments: disaggregation.

Cloud deployments disaggregate storage from compute, providing more flexibility to both the storage and compute layers. In this chapter, we explore disaggregation by taking it one step further and applying it to memory (DRAM). Disaggregated memory uses network attached DRAM as a way to decouple memory from CPU. In the context of databases, such a design offers significant advantages in terms of making a larger memory capacity available as a central pool to a collection of smaller processing nodes. To explore these possibilities, we have implemented Farview on top of Coyote. Farview is a disaggregated memory solution for databases that operates as a remote buffer cache with operator offloading capabilities.

Farview is an FPGA-based smart NIC making DRAM available as a disaggregated, network attached memory pool capable of performing data processing at line rate over data streams to/from disaggregated memory. Farview supports query offloading using operators such as selection, projection, aggregation, regular expression matching and encryption.

We evaluate Farview and demonstrate its competitiveness with a local buffer cache solution for all workloads. In several cases, it outperforms the local solution, confirming that a well-designed disaggregated memory can serve as a viable alternative for databases in cloud environments.

6.1 Motivation

Historically, databases have invested significant efforts to reduce I/O overheads. Initially, memory was very limited in capacity and disks were slow. Over time the bottleneck shifted as faster storage became available (SSDs, Non-Volatile Memory (NVM)), memories became larger, and multicore emerged. Yet, the I/O overhead remains a major factor in the overall performance. To minimize it, databases have relied on keeping more and more data in memory, a trend that cannot continue for two main reasons: databases induce considerable data movement, which is known to be highly inefficient in modern computing; and the amount of data to be processed keeps growing while DRAM capacity does not.

Optimized query plans typically push down selection and projection operators to filter out the base tables as early as possible. However, filtering base tables to get the data actually needed by the query is an expensive step. Base tables are fetched from storage as blocks that are placed in a buffer cache in memory. From there, a query thread reads the data and filters it to form the input to the rest of the query plan. Often, most of the data is dropped because it does not match the query's selection predicate. As more data is involved, the overhead becomes larger. Data movement has been identified as one of the biggest inefficiencies in computing [76, 77], making the way databases operate intrinsically problematic from a systems perspective, even if main memory could grow indefinitely.

However, DRAM capacity is also a major limitation, because the size of data processed by analytical databases keeps growing [131]. The current approach to tackle such a limitation is to use NVM, introduced as an alternative that is both cheaper and has higher capacity than DRAM (and persists data), but has larger latency. In databases, it is increasingly used to improve and expand the memory hierarchy [157, 177, 41, 40, 176, 42]. Such designs do not address the overhead of moving large data sets to the CPU, only to have most of it filtered or projected out. Specialized hardware between memory and the CPU has even been proposed to filter data as early as possible, minimizing bus congestion and cache pollution [25, 87].

An alternative approach for addressing memory pressure is to exploit the distributed nature of database engines, particularly in the cloud, to take advantage of non-local memory. In such distributed settings, the coupling of storage, compute, and memory capacity is problematic both cost-wise and performance-wise: the inability to independently provision each of those elements leads to inefficiencies due to over-provisioning. For instance, allocating large amounts of memory for tasks that are not compute-heavy leaves CPU unused, as other applications might not be

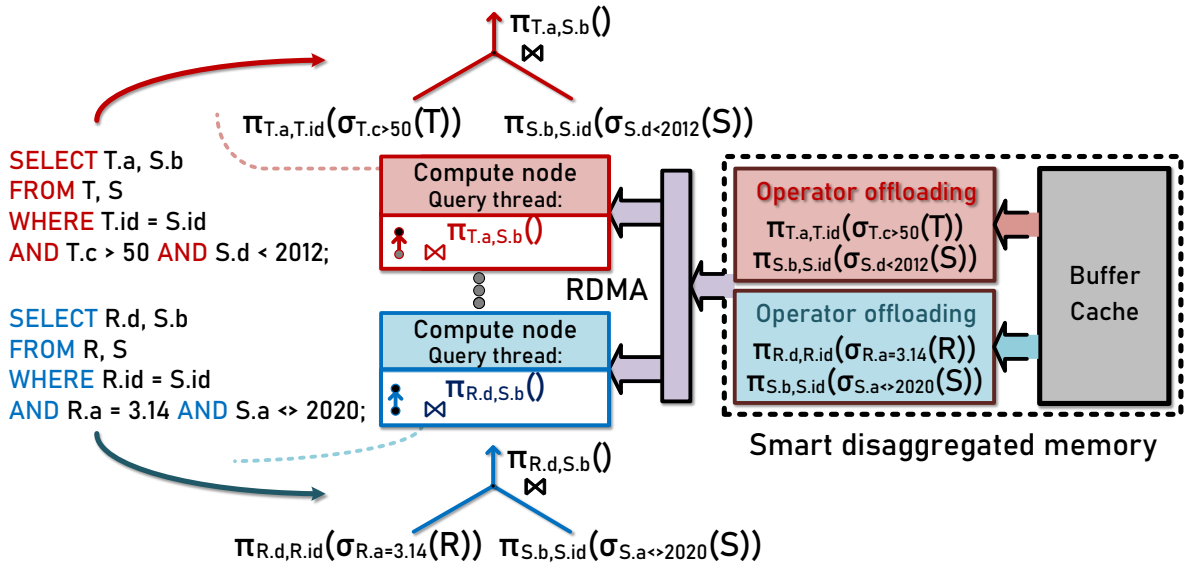

Figure 6.1: Farview query execution: Offloading of query operators to the smart disaggregated memory and splitting the query plan between compute and memory nodes.

able to run on the remaining memory. Conversely, allocating many virtual CPUs to a task may result in the memory being underutilized for lack of compute capacity for other tasks. Due to these challenges, essentially all cloud architectures follow a clear trend towards disaggregation. Currently, the most visible form of disaggregation is the separation of compute and storage. The next step is the disaggregation of memory and compute, which is being pursued in various forms: disaggregated DRAM [139, 94, 140], disaggregated persistent memory [147, 193], far memory [24, 29] and smart remote memory [187, 123].

With Farview we demonstrate that databases are uniquely positioned to exploit disaggregated memory to address both the issues of inefficient data movement and DRAM capacity. Our approach is based on physically detaching query processing from memory buffer management. The buffer pool is placed on network attached disaggregated memory, with query processing nodes provisioned on demand to run a query by reading the data from the network attached buffer pool. This design presents multiple advantages. Consider, for example, queries with low selectivity (e.g., TPC-H Q6) or an aggregation after a `GROUP BY` statement. In the first case, the query reads a large amount of data from the buffer pool just to discard most of it. In the second case, a query of the form `SELECT T.a, COUNT(*) FROM T GROUP BY T.a` will usually return only a handful of tuples, but it still requires reading the entire table. The smart disaggregated memory we propose offers the opportunity to (1) reduce data movement by pushing

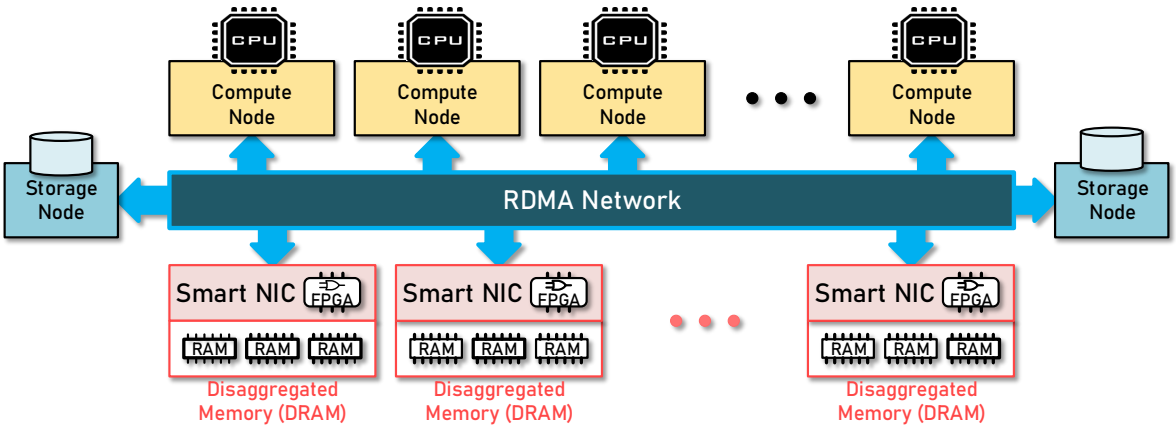

Figure 6.2: Disaggregation with Farview.

down operators to the disaggregated memory, so that the processing nodes receive only the relevant data; and (2) reduce memory requirements for computing nodes by centralizing the buffer cache in disaggregated memory and removing unnecessary copying of the data to the compute nodes. Figure 6.1 shows an example where projection and selection of two concurrent queries are offloaded to smart disaggregated memory, while the join and the final projection happen at the compute nodes.

To prove that these ideas can work in practice, we develop *Farview* (FV), a novel platform for data processing over disaggregated memory. Farview supports near-data processing to compensate for the added latency of accessing memory through the network by moving data reduction operators (selection, projection, aggregation, etc.) to the disaggregated memory.

Farview is an example of a system built on top of Coyote that makes comprehensive use of its features. In the context of Farview, Coyote is transformed into a dedicated smart NIC, which enables dynamic operator push-down on concurrent streams that read from memory. Farview effectively employs the RDMA stack, which has been extensively discussed in previous chapters, to optimize the interaction between the network and memory and minimize processing overhead on the CPU of the computing nodes, freeing up processing capacity.

From a database perspective, Farview functions as a disaggregated memory buffer pool with operator push-down capability. This memory is byte-addressable by the threads running queries on the computing nodes.

Farview currently supports a wide range of query operators: selection, projection, aggregation, distinct, group by, regular expression matching, and encryption. All these operators achieve near

line-rate speed, adding insignificant latency to baseline network overheads. Farview also supports concurrent access, with multiple clients all accessing the same disaggregated memory. Our experiments show that Farview induces almost no overhead over operating on local memory and provides significant performance gains when data can be reduced in the disaggregated memory.

In this work we focus on the design, architecture, and experimental evaluation of Farview when running queries, leaving other aspects such as cache replacement policies and query processing elasticity to future work.

6.2 Background and Related Work

In this section we motivate Farview and discuss related work. Farview is based on extensive experience in data center, computer, and processor design [55, 88]. For reasons of space, we focus here only on two salient aspects: memory disaggregation and near-data processing.

6.2.1 Coping With Memory Pressure

Data growth has turned DRAM into a major bottleneck [83, 131]. To cope with this bottleneck, advances in memory technologies and networking are leveraged to increase effective DRAM capacity. Within a local node, studies have explored compressing cold pages into local DRAM [131] or using local NVM directly as memory or with DRAM acting as a transparent caching layer [168, 171, 81]. These designs have also been used in databases in different ways, to expand virtual memory [157], directly as memory [40], or as a cache [176]. While in many cases there are performance advantages, these efforts require significant redesign in the database engine and do not address the underlying problem of inefficient data movement.

In a distributed setting, the notion that memory can be shared across a cluster of machines has been around for decades [36]. More recently, the advent of fast networks like InfiniBand FDR/EDR has renewed interest in exploiting memory (DRAM or NVM) accessed over the network. *Remote memory*, a distributed memory infrastructure where a group of comparably equipped compute nodes make their memory available to their peers, has been exposed to applications as a remote paging device [29, 102], as a file system [22], and as distributed shared memory [79, 155, 182]. Although this organization leverages existing resources and can improve resource utilization of otherwise unused memory, it entangles compute and memory for provisioning and expands the failure domain and attack surfaces of each machine [131, 23].

In contrast, *disaggregated memory* systems use network attached memory that is distinct from the memory in the compute nodes [140, 193]. This approach allows the disaggregated memory to scale independently of the system's computing or storage capacity [139], and removes the need to over-provision one resource to scale another. From the database perspective, this is a promising architecture. An evaluation of existing database engines (MonetDB and PostgresSQL) using LegOS [181], an operating system for disaggregated memory, indicates that the network overhead is the main bottleneck [213, 214]. The authors conclude that disaggregated memory has potential, but significant performance loss occurs due to the use of sub-optimal algorithms and lack of suitable data structures.

In Farview, we demonstrate that disaggregated memory is especially suitable to database engines when used as a buffer pool (also suggested in [213]). This makes the integration of disaggregated memory a more natural way to address the memory capacity limitation as neither the interface to memory needs to be changed nor the memory hierarchy expanded. What remains to be addressed are the data movement inefficiencies and network overheads.

6.2.2 Efficient Data Movement

Data movement inefficiencies can be addressed by using near-data processing. Expanding on decades-old work that memory and storage can be active components [163, 121, 170, 169], several approaches to memory disaggregation explore increasing the intelligence of network-attached memory. *Far memory* [24] proposes simple hardware extensions to reduce the number of network traversals to access non-local memory, and support for efficient notifications to facilitate consistency of data cached in the local memory of the nodes. In the context of databases, the advantages of processing data in the disaggregated memory have also been suggested [213, 214], but without proposing a possible implementation. The argument in favor of such designs is simple: push down selection and projection operators (as well as potentially other operators such as grouping, aggregation or even joins where one of the tables is small) to the memory or storage so that the base table is filtered out in-situ and irrelevant data does not need to be moved or sent. Although to our knowledge not yet used with disaggregated memory, the idea mirrors a growing trend to push SQL operators near the data, until now mostly to storage [205, 113]. Even more ambitious are accelerators embedded in the data path between memory and CPU caches [71, 87], which can filter data as it is read from memory to reduce data movement and cache pollution. Finally, in the cloud, systems like Amazon's AQUA [45] use SSDs attached to FPGAs to implement a caching layer for RedShift that supports SQL filtering operations and operator

push-down to minimize the amount of data movement from storage to the processing nodes. These designs are based on introducing a *bump-in-the-wire* processor to be able to process data closer to where it initially resides, instead of moving it first and then processing it. In Farview we adopt a similar design, with operators placed directly in the path of the network and memory. Farview requires neither changes to the storage layer interface nor specialized processors, whilst adding a layer of dynamicity provided by the reconfigurable platforms. Moreover, we focus on disaggregating the buffer pool in DRAM, rather than introducing additional caching layers between storage and compute.

The network overheads can be addressed using advances in networking (in addition to compensating for it by using near-data processing). Most of the work on different forms of disaggregated memory utilizes low-latency RDMA instead of TCP/IP, often extending one-sided operations on RDMA to offload group-based operations for storage replication (e.g., HyperLoop [123]), concurrency and transactions for data structures (e.g., AsymNVM [147]), and memory access operations for key-value stores (e.g., StRoM [187]). RDMA employs the network protocol (InfiniBand [107], RoCE [108]) and the Network Interface Card (NIC) to move data directly between the memory of different machines. At the speed at which networks operate today, RDMA can be used to efficiently transfer large amounts of data across machines at the rates of DRAM memory channels [93]. It is thus especially suitable for disaggregated memory and databases [27]. It has been shown to speed up distributed operators such as data shuffling [143], joins [48, 49], transactional workloads [50], and indexing [215]. In Farview, we use RDMA to efficiently transfer data through the network so that the query processing thread directly gets the data from the remote buffer pool. As suggested by current architectural trends in the cloud, Farview is implemented on top of Coyote and functions as an FPGA-based smart NIC. It supports SQL operators acting on the RDMA data streams as they move along the data path connecting the disaggregated memory to the network. The design efficiently combines near-data processing with faster network transfers, while removing the need for a conventional CPU to support the disaggregated memory (a design that resembles that of AQUA, which also uses FPGAs instead of conventional CPUs, and that is aligned with how FPGAs are deployed in Microsoft's Azure [172, 62, 91]).

6.3 System Overview

Farview is a smart disaggregated memory attached to the smart NIC with operator offloading capabilities that behaves as a database buffer pool. Traditionally, query processing threads access base tables by reading them from the buffer pool and copying the data to their private working

space. With Farview, nothing changes for the query thread, except that the read operation is on a remote disaggregated memory rather than local memory, potentially with a subset of the operators already applied.

6.3.1 Smart Buffer Pool With Operator Offloading

Farview exposes a data API to the buffer pool (Section 6.4.2) that can offload operators to the disaggregated memory. Farview executes an *operator pipeline* with one or more operators (e.g., a selection and then an aggregation) to process the data as it is read from disaggregated memory, effectively functioning as a bump-in-the-wire stream processor (Section 6.5). As done in conventional query processing, operator pipelines are constructed from individual blocks that implement a given operator and provide standard interfaces to combine them into pipelines. The modular nature of these pipelines and the reconfigurability of the FPGAs allows the list of operators to be swapped and easily extended in the future (e.g., join operators).

Farview currently supports a range of operators, including:

1. **Projection** operators to reduce the columns returned (and potentially reduce memory accesses).
2. **Selection** operators that filter data according to a collection of predicates.
3. **Grouping** operators that combine tuples (e.g., distinct, group by and aggregation).
4. **System support** operators that process data in-situ before sending the data (e.g., encryption/decryption) and perform system optimization tasks like packing the data to reduce the overall network usage.

6.3.2 FPGA-Based Architecture

Prototyping smart disaggregated memory requires several components, including DRAM, memory controllers, a network stack, a mechanism to support concurrent access to the memory, and stream processing capacity for operator push-down. Modern FPGAs are a natural match for such functionality, as high performance and flexibility can all be combined in a single device rather than having to connect separate components such as processor, NIC, and memory, all inducing significant data movement overheads. FPGAs can also support substantial amounts of

local memory, directly attached to the FPGA. This on-board memory is usually organized in multiple channels (even High Bandwidth Memory (HBM)) [70]. Farview's design (Section 6.4) leverages these characteristics and abstractions provided by Coyote to implement disaggregated memory as a lean component.

To deploy operators that can process data on the disaggregated memory, the FPGA is divided into multiple isolated *virtual FPGAs* that operate concurrently. These dynamic regions can be obtained by different clients and can process different query requests. Each dynamic region serves an access request to the disaggregated memory and can implement a separate operator pipeline that can execute a set of different queries.

In this context, we make use of Coyote's dynamic reconfiguration capabilities. The logic deployed in the vFPGAs can be swapped at runtime, eliminating the need to reconfigure the entire FPGA. As demonstrated earlier, this swapping process takes a matter of milliseconds, with the exact time dependent on the size of the region being swapped.

A combination of operators is precompiled as an operator pipeline into a hardware design that is dynamically loaded into the FPGA at runtime, upon a request from a client (i.e., a thread processing a query at a computing node). The operators and their pipelines can be modified or extended, and new ones can easily be added by combining the existing ones or changing their parameters.

6.4 Implementation

Farview is implemented on top of Coyote. The primary abstractions we leverage from Coyote include FPGA-side services such as the RDMA network stack, memory stack, memory virtualization and dynamic reconfiguration.

6.4.1 Architecture

Farview is organized around three main modules: the network stack, the memory stack, and the operator stack. A closer examination of this architecture (Figure 6.3) reveals the absence of the host CPU, which represents a significant difference in Farview compared to the base Coyote system. This is due to the fact that data paths no longer require access to the host memory. Instead, the path between the disaggregated memory and RDMA serves as the only data path in the system. As Farview is a research platform, we still use the host CPU to ease management

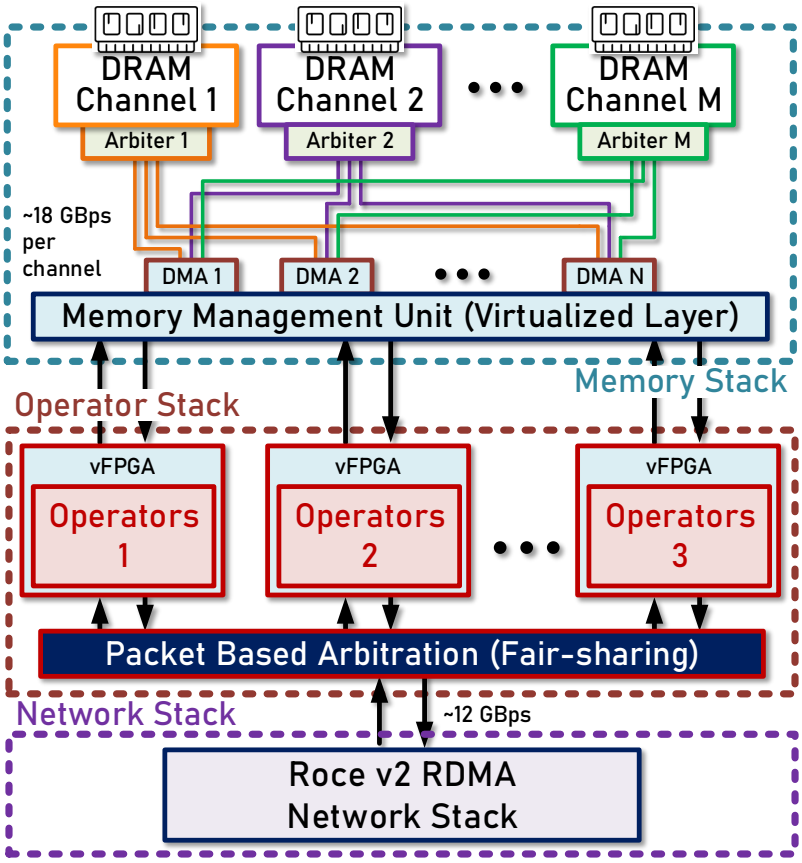

Figure 6.3: High level view of Farview's architecture.

operations. However, in practical deployment scenarios, these management operations could potentially be handled by a softcore CPU within the FPGA fabric. Besides reducing the system's cost, one possible additional benefit of this potential architecture is the reduction of FPGA-side resources, as the host-side interconnect would not be needed.

The network stack manages all external connections and RDMA requests, providing fair share mechanisms across all concurrent accesses. The memory stack implements the buffer pool, and can be used as regular memory, with blocks/pages being loaded from storage as needed. The memory stack houses the memory management unit (MMU), which handles all address translations to the on-board memory attached to the FPGA and provides the necessary arbitration and isolation between concurrent accesses. The operator stack contains the dynamic logic necessary to push down operators to the disaggregated memory. It lies between the memory and the network stacks and can be seen as a specialized stream processor acting on the data as it moves from the memory stack to the network stack. It also controls how data is retrieved from the

memory stack. The operator stack is reconfigurable, and its operator logic can be changed at any point without affecting the operation of the rest of the system.

Clients access the disaggregated memory by opening a connection with Farview to one of the dynamic regions, each able to contain one of the possible operator pipelines. Whenever a client makes a request to Farview, the network stack routes the request to the correct virtual dynamic region in the operator stack belonging to the client that initiated the request. The read request is forwarded to the memory stack, which translates the virtual address to a physical address in the on-board FPGA memory, and then issues the actual data request. The clients have the local catalog information that is used to determine the addresses of the tables to be accessed. The returned data is streamed back to the dynamic region, where the loaded operators are applied. Finally, the resulting data is forwarded to the network stack, and further sent directly to the memory of the client.

The interfaces of the data and the requests are all based on a simple *AXI stream* handshaking protocol [145], which provides uncomplicated synchronization, pipelining, and backpressure mechanisms, all allowing Farview nodes to support processing at high throughput. The standard interfaces help with portability across different boards. The dynamic region where the operators are loaded always exposes the same set of interfaces to the operators, thereby simplifying the task of creating the operators. This protocol also permits deep pipelining of the overall design, which allows processing to occur simultaneously in different areas of the system.

To attain high frequencies and reduce the impact of the physical distance between the stacks, data is buffered in queues as it traverses from one stack to the other. The queues, as well as any temporary state created by Farview operators, are implemented using fast on-chip FPGA memory. The buffering allows clear decoupling of processing stages, which helps with structuring the overall system and allows Farview nodes to achieve the operating frequencies necessary to sustain processing at line speeds. The frequencies of the components in Farview range between 250 MHz (network stack, operator stack) and 300 MHz (memory stack).

Compared to existing FPGA frameworks, in Farview the dynamic regions must be connected to network and memory stacks, which have fixed locations within the FPGA, thus reducing the degrees of freedom in placement and sizing. Farview's management infrastructure must cope with network speeds of 100 Gbps (and even higher internal speeds), which require wide buses (512 bit) [187, 179], further restricting region placement and sizing. We choose predefined dynamic regions to accommodate these placement and sizing restrictions. In practice this implies that the size of each virtual dynamic region is fixed and cannot be changed. However, each region is more than large enough for the purposes of offloading the operators we intend to support.

6.4.2 Farview Programmatic Interface

Farview exposes a simple high level data API covering both the critical path operations and connection management operations. The former utilize the high performance 100G fabric, whilst the latter are handled via regular TCP/IP connections. The critical path operations include both standard low level one-sided RDMA read and write commands to read (write) data from (to) memory and an extra Farview command, which invokes the operator(s) in the Farview node directly over the read data stream. We use the Farview command as the basis for more complex SQL expressions.

A client running on a remote computing node begins by establishing a connection to Farview. In response, it gets a created object representing the connection (*QPair*), which holds all the necessary information for the connection and is used as an argument to subsequent Farview methods. The following function is used for this purpose:

```
bool openConnection(QPair *qp, FView *node);
```

The client can then at any point request that a different operator pipeline be loaded. Operator pipelines are precompiled and kept in the operator pipeline library on the target Farview node. The operation is executed with the following command, where the *opid* is a unique id of the requested pipeline:

```
bool loadPipeline(QPair *qp, int32_t opid);
```

Farview memory is virtualized, and internal management is handled by the MMU in the Farview node. Since we are focusing on read-only scenarios, Farview does not currently provide concurrency control. Clients allocate memory for tables using the following allocation functions:

```
bool allocTableMem(QPair* qp, FTable *ft);
void freeTableMem(QPair *qp, FTable *ft);
```

Regular RDMA requests for simple reading/writing of the remote table can be sent with the following two functions:

```
void tableRead(QPair* qp, FTable *ft);
void tableWrite(QPair* qp, FTable *ft);
```

Farview's request corresponds to a specialized InfiniBand verb [107] that invokes the remote processing capabilities with an arbitrary parameter set, specific to each operator pipeline. This verb is invoked in the following generic function, which is used as the basis for building additional higher level functions supporting specific operator combinations and queries:

```
void farView(QPair* qp, FTable *ft, uint64_t *params);
```

As an example higher level function, we present a selection operator with real number predicates:

```
void select(QPair* qp, FTable *ft, uint64_t *projection_flags,
  ↪ uint64_t *selection_flags, float predicate);
```

This function can be used for the following type of queries:

```
SELECT S.a FROM S WHERE S.c > 3.14;
```

In this case, the `projection_flags` variable signals column `a` while `selection_flags` signals column `c`. The predicate is passed as a value. More complex variations are possible: for instance, if the hardware operator supports it, the predicate operand could also be a variable.

The interface presented here is intended to be used by the frontend query engine in Farview, rather than directly by the client.

6.4.3 Network and Memory Stacks

Coyote provides network and memory stack services that ensure efficient interleaving, fair-sharing of the resources, and robust security for stable operations. In the context of Farview, the network stack is responsible for managing all client to server connections and communcation, while the disaggregated buffer pool is implemented in the memory stack using the on-board DRAM or HBM memory attached to the FPGA.

Upon connection establishment, each network connection flow and its corresponding queue pair are associated with one of the available vFPGAs and in turn with data streams from the Farview memory stack. These data streams are used to read the queried data into the vFPGAs, process it, and send it over the network. The flow is ready to process data once the operator pipeline corresponding to the flow is present in the vFPGA.

The RDMA queue pairs keep track of memory buffers both in the server nodes (where the tables reside) and the client nodes (where results are written). As part of establishing the connection flow, the client sends the virtual address of its local buffer to the server Farview node. Upon completion of query requests, the results will be loaded into this client buffer.

The buffer in the server Farview node can also be shared between different queue pairs, enabling multiple requests to share intermediate results. The current version of Farview focuses on read-only operators and has no transactional memory support.

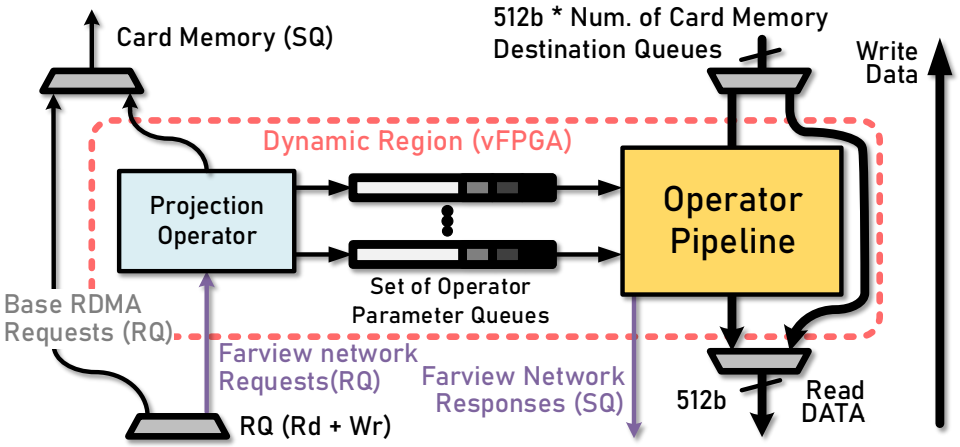

Figure 6.4: Single dynamic region (vFPGA) in the operator stack.

The method to execute queries in Farview (initiated by client nodes) involves a combination of two communication actions: the two-sided `SEND` verb, utilized to invoke the query, and the one-sided `RDMA WRITE` verb in the opposite direction, employed to transmit the query results back to the client. A detailed explanation of this operation has been covered in Section 4.4.5.

Requests sent using the `SEND` verb are directly written into the receive queues within the vFPGAs, bypassing the intermediate step of storing them in memory. This allows operators to promptly respond to these requests without incurring any additional latency overhead. The `SEND` verb can also include a variable number of operator-specific parameters, which serve to instruct the disaggregated memory on how to access and handle the data effectively.

6.4.4 Operator Stack

The operator stack is where the operator pipelines attached to connection flows to/from memory are deployed. The operator stack is implemented as a collection of predefined dynamic regions (vFPGAs). The operators deployed in the dynamic regions use the interfaces exposed by Farview's network and memory stacks.

Operator pipeline logic can be deployed and swapped on-the-fly without affecting the integrity and operation of the system or other operator pipelines belonging to other clients. These regions and their access to memory are isolated from each other. Such functionality is typically not available in commercial systems, but very often studied in the literature, e.g., [122, 210].

Figure 6.4 illustrates how a single dynamic region processes a query request. The base RDMA read and write requests forward the virtual address and transfer length parameters directly to the memory stack and to the MMU, bypassing the dynamic region. If the request is a simple RDMA read/write request, it contains no additional parameters. If the request is a Farview command, it carries a number of operator-specific parameters (purple path, receive queue), along with information about the virtual memory locations it is accessing. The number of parameters can vary depending on the specific operators that are present in the operator pipeline.

The write path allows RDMA updates to the memory. The operators' bump-in-the-wire data processing occurs along the read data path. The width of the data path's input to the dynamic region scales with the number of available memory channels (destination queues). As a result, with the aforementioned striping technique, each dynamic region gets the full bandwidth potential of the disaggregated memory and its multiple memory channels. The output is forwarded to the network stack using a 64-byte datapath width, the same as the provided network interface.

Query responses are sent via the response channel (purple path, send queue), which initiates one-sided RDMA transfers to the client. The operator pipeline dynamically generates these responses for each result packet only when the packet is ready to be sent to the client node. This approach enables the operators to dynamically control the size of the result data transfers, which is important for operators (e.g., filtering) where the size of the result data is unknown when the request arrives, prior to processing. The direct data streams between the memory controller, the operator, and the network are scaled so as to saturate the bandwidth in each module and to provide optimal performance.

6.5 Operators and Pipelines

In this section we discuss operator pipelines and four classes of operators: projection, selection, grouping and system support.

6.5.1 Operator Pipelines

As described above, a query is transformed into an operator pipeline, which is deployed on a dynamic region allocated to the corresponding client. An operator pipeline contains one or more operators that provide partial query processing on datapath operations to disaggregated mem-

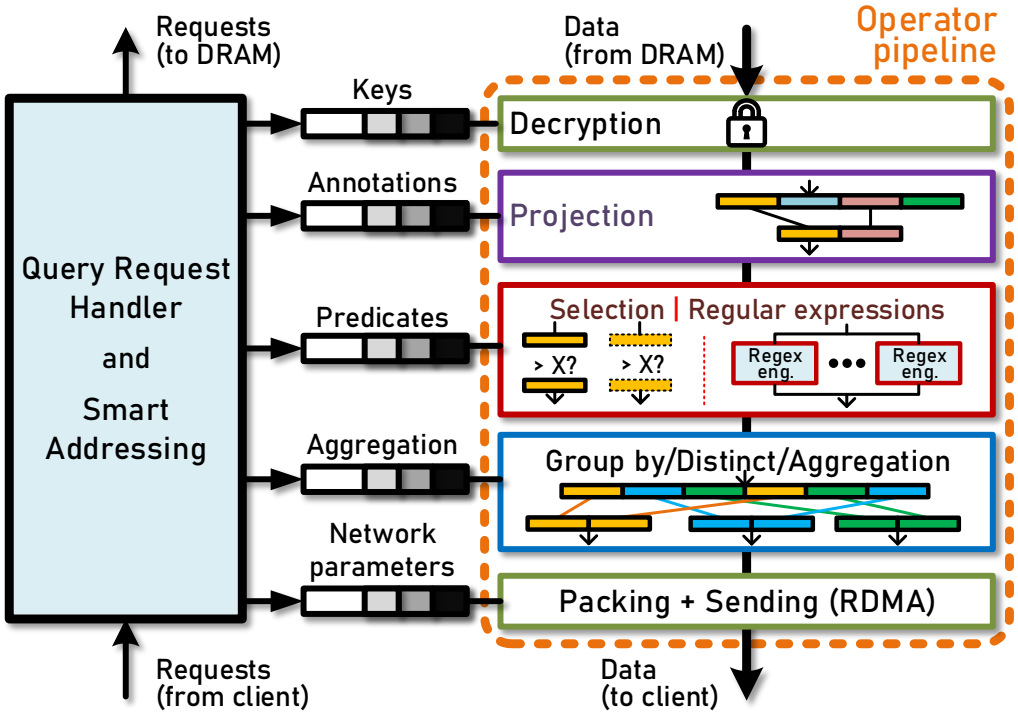

Figure 6.5: Operator pipeline example.

ory. This processing is effectively a bump-in-the-wire that operates on data without introducing significant overheads.

Figure 6.5 illustrates a generic operator pipeline that includes a broad set of operator classes, including projection, selection (e.g., predicate selection, regular expression matching), grouping (e.g., distinct, group by, and aggregation), and system support (e.g., encryption/decryption).[1] These example operators are described in more detail in the remainder of this section. Which operators are actually present in the pipeline depends on the requested set of queries to be executed. In one scenario, the pipeline can support projection, followed by selection and group by. In another, it can support regular expression matching on encrypted strings, which requires decryption early in the pipeline. The reconfigurable nature of the regions provides flexibility, as it allows arbitrary operator types and combinations to be natively supported by the system.

[1]We assume that all data is stored in row format, but there is nothing intrinsic preventing support for column data.

When a query request arrives, it is first forwarded to the query request handler, which requests the data from the memory stack via smart addressing. At the same time, any necessary parameters for additional processing are forwarded to the remaining operators in the pipeline. Data arriving from the memory is processed in a streaming fashion by these operators. Once the processing is done, the resulting data is sent back to the client over the network. Each pipeline has the potential to be fed with input data each FPGA clock cycle. In query terms, this translates to each pipeline being fed with up to a single tuple in each cycle. In the same manner, a pipeline has the potential to produce results on the output of every cycle. Using this design, operator processing overhead can be efficiently hidden behind the memory and network operations.

Operators are written by the Farview developer as part of the smart disaggregated memory system design, using common hardware description languages like VHDL or Verilog or in the C++-like syntax supported by high-level synthesis tools such as Vivado HLS. The operator pipeline is precompiled, so that it can be deployed to a dynamic region at runtime. Operator implementations use Farview's network, memory and operator stack interfaces, rather than the interfaces of the underlying FPGA board, which makes the operators portable across Farview deployments on different platforms.

6.5.2 Projection Operators

Projection: A common operation in databases is projection, which returns a subset of a table's columns. Consider for example a projection of the form `SELECT S.car, S.price FROM S`, where `S.car` and `S.price` are non-consecutive attributes and have a number of fixed-length attributes between them. The projection operator reads the table from the disaggregated memory, parses the incoming data stream based on query parameters describing the tuples and their size, and projects the requested tuples into the pipeline for further processing using annotation. During parsing, the tuples are annotated with parameters from the requested query, and obtained from the parameter queues. These parameters are simple flags that state which of the columns are part of the selection, projection or grouping phases. Their interpretation depends on the actual combination of operators being used and their specific implementations.

Smart addressing: In scenarios where queries request only a small subset of the columns from a very wide table, performance would benefit from reading only the requested columns from memory, rather than reading full rows and applying the projection on the incoming data stream. For this purpose, we implement a smart addressing optimization that issues multiple, more specific, data requests to memory. Smart addressing is most effective when the total number of

columns per tuple is large and the number of projected columns is much smaller than the total; otherwise, it is more efficient to read entire tuples and project using annotations, as described above, since the memory access is sequential. We explore the crossover point between these two modes in Section 6.6.2.

6.5.3 Selection Operators

Selection operations that filter data directly map to the SQL `WHERE` clause (e.g., in queries of the form `SELECT * FROM T WHERE T.a > 50`). These operators have the ability to greatly reduce the amount of data to be processed by later stages and ultimately the overall amount of data sent through the network, thus reducing the overall network bandwidth usage. For example in TPC-H Q6, only 2% of the data is finally selected. Pushing the filtering to the disaggregated memory reduces the I/O overhead by orders of magnitude. Farview's selection operators consist of predicate selection, regular expression matching and vectorized selection operators.

Predicate selection: Farview implements a versatile predicate selection pipeline (Figure 6.6) capable of handling a wide array of predicate combinations for evaluation. Within a tuple, the selection pipeline can compare an attribute's value to a constant specified in the query request. Simultaneously, it provides the capability to directly compare different attributes within the same tuple. This is achieved through configurable comparator blocks, which can be tailored using the query request. The results of these comparisons are subsequently directed to a truth table, implementing the predicate evaluation function supplied in the query request. This approach enables the implementation of a highly adaptable selection pipeline, minimizing the need for dynamic reconfigurations, as it supports a diverse range of queries (while incurring minimal increase in resource consumption).

Regular expression matching: String matching is becoming an increasingly important operator in SQL (e.g., using either `LIKE` predicates as in TPC-H Q16 or regular expression matching). It is even more important in unstructured data types, such as in the case of JSON fields in PostgreSQL. In Farview we have integrated an open source regular expression library for FPGAs [111] and use it to filter strings. In these operators, data is retrieved from the remote node only when it matches the given regular expression. The operator implements regular expression matching using multiple parallel engines, instantiated in the operator stack. The parallelization allows the module to fully sustain processing at line rate. Unlike software solutions, the performance of the operator is dominated by the length of the string and does not depend on the complexity of the regular expression used [111].

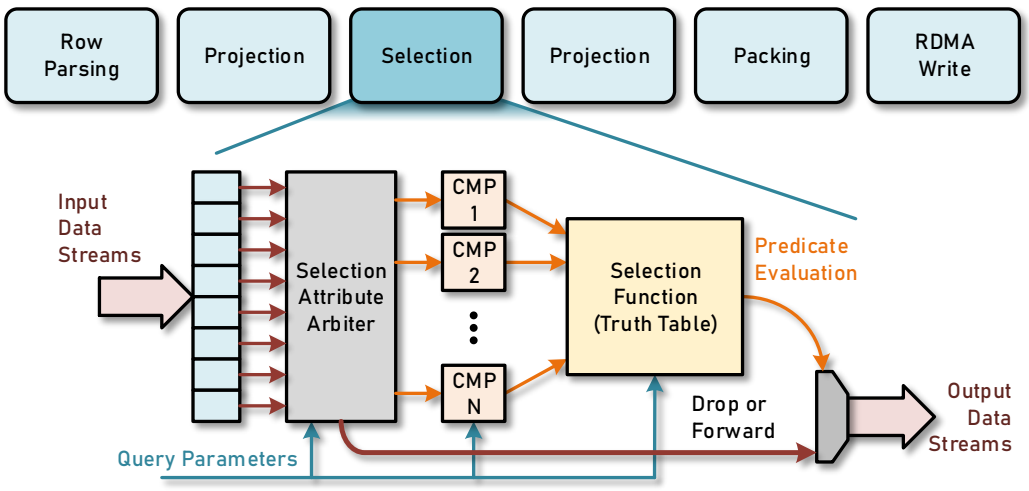

Figure 6.6: Generic predicate selection pipeline in Farview.

Vectorization: Farview implements a limited form of vectorization as an optimization to improve the performance of stateless operators like selection. To alleviate the inefficiencies of the tuple-at-a-time query processing model [99] and its `next()` function calls that pass tuples from one operator to the next, the database community has adopted query compilation [156] and column stores. Column stores either process data in full batches like MonetDB [54], or in smaller vectors like VectorWise [217]. The latter allows the use of tight loops and/or SIMD instructions to process column data, allowing DBMSs to take advantage of the latest CPU advances for data processing. In Farview, we use a vectorized model similar to that of VectorWise, but with a vector size that is chosen based on the degree of memory striping (described above), rather than trying to fit the size of the processor's L1 cache. With vectorization, data is read in parallel from multiple memory channels, and individual tuples are emitted to a set of selection operators executing in parallel. The number of parallel operators is chosen based on the number of memory channels and the tuple width. This approach achieves both higher read bandwidth from the memory stack (due to memory striping) and higher processing throughput (due to the parallel operators). At the moment the operator pipeline with vectorization is enabled for the simpler queries and workloads that can be easily parallelized without data dependencies (e.g., selection).

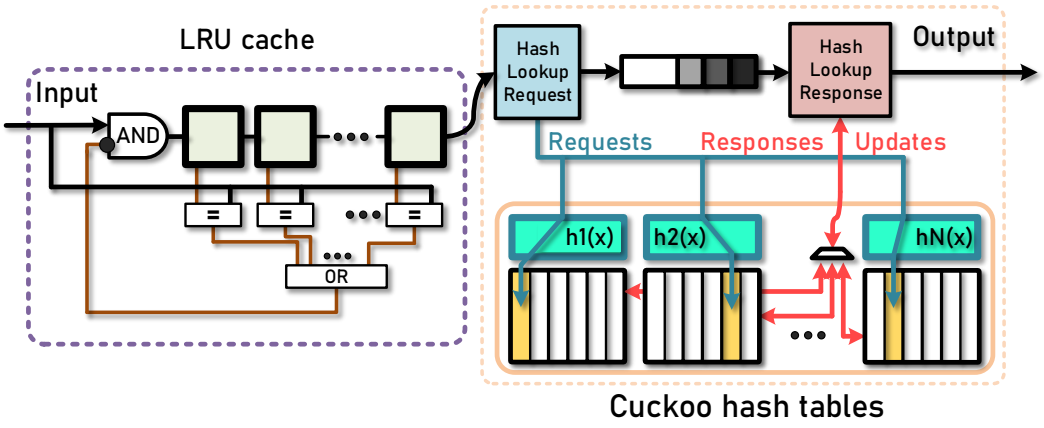

Figure 6.7: Architecture of the aggregation operator.

6.5.4 Grouping Operators

Distinct: The distinct operator eliminates repeated column entries before they are sent over the network. It directly maps to the SQL `DISTINCT` clause, in queries such as: `SELECT DISTINCT T.a, T.b FROM T`. It operates by hashing the values and preserving the entries in the hash tables present in the fast FPGA on-chip memory. As the complete hashing is calculated in the FPGA, the distinct operation can be done on multiple columns without noticeable performance overhead, but using more FPGA resources.

To sustain the line rate without negatively affecting the overall pipeline processing, the distinct operator needs to be fully pipelined in order to overcome the latency of the lookups and updates of the hash table. This pipelining creates potential data hazards, in the case where two successive tuples with the same value will be inserted into the hash table and ultimately sent over the network as distinct elements. Because of the latency of the hash table, the second (following) tuple cannot see the update produced by the first one. To approach this problem we apply the strategy explained in [205] by implementing an LRU cache to hide the hash table latency. The main difference is the far higher line rate that we have to sustain in our system (over 40 times greater), yielding additional design constraints.

To guarantee full pipelining and constant lookup times, the hash table that we implement does not handle collisions. Instead, collisions are written into a buffer, which is sent to the client to be deduplicated in software. To greatly reduce the collision likelihood, we implement cuckoo hashing, with several hash tables that can be looked up in parallel. Upon eviction from one of the tables, the evicted entry is inserted into the next hash table with a different function. This

occurs in the background and does not affect the full pipelining of the operator.

To successfully hide the latency of the hash table, we implement a cache to hold the most recent keys. The cache needs to be a true Least Recently Used (LRU) cache in order to guarantee protection from possible data hazards. The standard implementations of LRU caches come with a lot of overhead, as pointers and extra history-keeping data structures need to be present. For this reason, we implement the cache with a shift register, which adds a negligible latency to the data streams (the amount depends on the number of cuckoo hash tables), but is able to efficiently provide a quick lookup. The nature of the shift register provides a true LRU replacement policy and this solution thus fully satisfies the strict requirements imposed by Farview. The architecture of the distinct operator (and other aggregation operators) is shown in Figure 6.7.

Group by: In many applications, data is read and grouped to perform some form of aggregation (e.g., TPC-H Q1). Operations like these directly map to the SQL `GROUP BY` clause (e.g., queries such as `SELECT T.b, COUNT(*) FROM T GROUP BY T.b`).

Farview provides a group by operator with a structure similar to the distinct operator, with analogous challenges and design choices. The same cuckoo hash tables are used to preserve the groups. The implemented cache in this case is write-through, as it is no longer sufficient to just discard the data prior to sending it. The operator reads the complete table and all of its tuples without sending anything over the network, to perform the full aggregation. At the same time, it inserts the distinct entries into a separate queue. Once the aggregation has completed, the queue is used to lookup and flush the entries from the hash table along with any of the requested aggregation results to the network.

Aggregation: Aggregation operators can easily be supported in FPGAs as standalone, where simple computations are performed directly on the passing data streams, or on top of the group by operator, where each entry in the hash table contains an additional aggregation result. Farview supports a range of standard aggregation operators like `count`, `min`, `max`, `sum` and `average`.

6.5.5 System Support Operators

Encryption/Decryption: A key concern for both remote memory and smart disaggregated memory is the need for data encryption [131]. Farview implements encryption so that the data is treated similarly to Microsoft's Cypherbase [38], where a database stores only encrypted data, but can still answer queries over such data by using an FPGA as a trusted module. We have implemented encryption as an operator using 128-bit AES in counter mode. Since the AES

module is fully parallelized and pipelined, it can operate at full network bandwidth. This means that no throughput penalty is paid when this operator is applied on the stream, incurring only a negligible overhead in latency. This allows the data to be stored securely. Because no real processing penalty is incurred, encryption/decryption can be placed at both the input and the output of the operator pipeline. Similarly one could provide additional system support operators such as compression, decompression, etc.

Packing: At the end of the processing pipeline, the annotated columns are first packed based on their annotation flags in a bid to reduce the overall data sent over the network. Multiple columns across the tuples are packed into 64 byte words prior to being written into the output queue. This packing uses an overflow buffer to efficiently sustain the line rate. In case of the vectorized processing model, the tuples are first combined from each of the parallel pipelines with a simple round-robin arbiter.

Sending: The sender unit is the final step before the results are emitted to the network stack. It monitors the queue where the packed results are written, and, based on the queue's status, issues specific RDMA packet commands needed to produce correct packet header information in the network stack. The sender module's dynamic approach to handling RDMA commands allows us to create RDMA commands even when the final data size is not known a priori, as is the case with most of the operators.

6.6 Evaluation

In this section we evaluate Farview's performance, and compare it with alternatives using a local buffer cache or a remote buffer cache. We first describe our experimental setup, including the hardware implementation details of our platforms. We then measure individual query performance using various operators, and query performance with multiple clients.

6.6.1 Experimental Setup

We compare Farview's smart disaggregated buffer pool (**FV**) with two different baselines: a buffer cache implemented in local (client) memory, where the processing is done on the local CPU (**LCPU**), and a remote buffer cache implemented on the memory of a different machine and reachable through a commercial NIC via two-sided RDMA operations (**RCPU**). This latter configuration resembles what is being done today for storage, where part of the processing is

Table 6.1: Resource consumption of Farview

Configuration	**CLB LUTs**	**Regs**	**BRAM tiles**	**DSPs**
6 regions	29%	28%	30%	0%
Operators (per dynamic region)	**CLB LUTs**	**Regs**	**BRAM tiles**	**DSPs**
Projection/Selection/Aggregation	2.7%	1.4%	0%	0%
Regular expression	2.3%	<1%	0%	0%
Distinct/Group by	2.1%	1.3%	8%	0%
En(de)cryption	3.6%	<1%	0%	0%
Packing/Sending	<1%	<1%	0%	0%

moved to a CPU located in the storage server. It also matches the definition of remote memory proposed in the literature. For RDMA microbenchmark experiments, we compare remote reads from Farview (**FV**) to remote reads to a different machine using one-sided RDMA operations over a commercial NIC (**RNIC**) that accesses the remote memory over PCIe. Finally, in the selection tests we also use a version of Farview with vectorization (**FV-V**).

For these tests, Farview runs on an Alveo-u250 board [9]. The board has up to 4 DRAM channels (16GB each) connected directly to the FPGA. Each channel has a softcore (running as reconfigurable logic on the FPGA) DRAM controller with a maximum theoretical bandwidth of 18GB/s. In our tests we used two of the four available channels to reduce compilation times. The tests can be done with four channels as well, which might yield some additional performance in specific cases. The board has two (QSFP) 100Gbps network ports. The boards are directly connected to each other through one of the two available QSFP (or network) ports.

The CPU baselines contain Intel Xeon Gold CPUs: **LCPU** uses a Xeon 6248 (clocked at 3.0–3.7 GHz), and **RCPU** uses a Xeon 6154 (clocked at 2.5–3.9 GHz). For the remote buffer cache and one-sided RDMA microbenchmark baselines (**RCPU** and **RNIC**, respectively), we used a commercial Mellanox 100G card (ConnectX-5 VPI) [189]. For the CPU-based baselines we used all available compiler and code optimizations.

As shown in Table 6.1, Farview requires doesn't require large amounts of FPGA resources (around 30% of the total on-chip resources) to implement the operator stack, the network stack, and the memory stack. The majority of the utilized on-chip memory is attributed to the memory management unit and the state keeping structures of the operator and network stack. Most of the implemented operators are not compute heavy and do not consume many resources, making it easier to combine them.

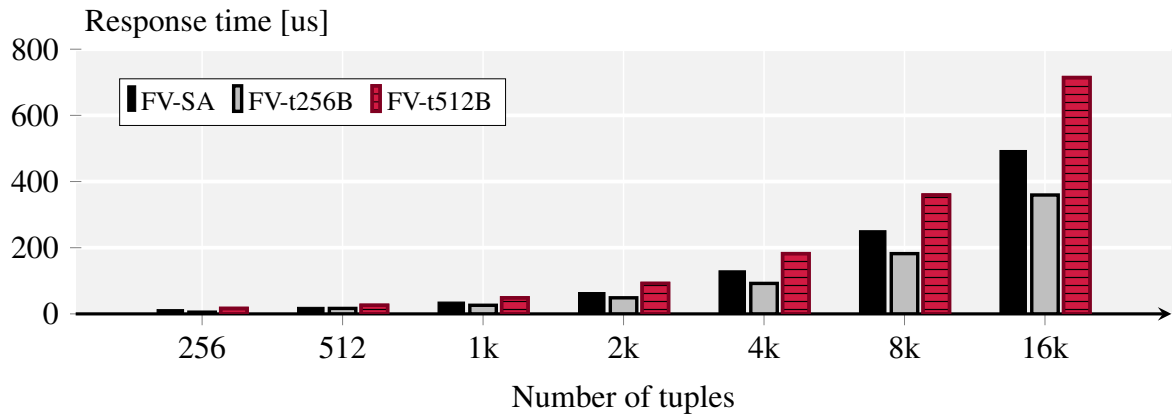

Figure 6.8: Standard projection vs. smart addressing.

For each experiment, we measure the running time until the final results are written to the memory of the client machine for both Farview and the baselines. This makes the performance numbers comparable, as the initial and end states are the same. We evaluate performance for a range of the operators and supported queries. Unless otherwise mentioned, our base tables consist of 8 attributes, where each attribute is 8 bytes long. The results for each experiment are averaged over multiple runs. As the CPU configurations may experience interference (e.g., due to context switches, interrupts), the **LCPU** and **RCPU** results are averaged over 10000 runs. Because the FPGA circuits provide more deterministic behavior, the Farview (**FV**) experiments are averaged over 1000 runs.

6.6.2 Projection

To investigate how to maximize the efficiency of accessing DRAM, we compare the standard projection operator, where the whole table is fetched from memory and projection is done in the first pipeline stage, versus the smart addressing operator, which issues individual memory requests only for the target projected columns. In this experiment, we project three contiguous 8-byte columns from a larger row. Figure 6.8 illustrates execution times for the smart addressing operator on a 512-byte tuple (**FV-SA**) and the standard projection operator with tuple sizes of 256 bytes and 512 bytes (**FV-t256B** and **FV-t512B**, respectively). For smaller tuples (**FV-t256B**), it is more beneficial to read the whole table sequentially from the DRAM memory and handle the projection in the operator pipeline. Once the tuples become larger (512 bytes), it is better to use smart addressing to read only the columns that are required by the query (**FV-SA**). In the following experiments, which use 64B tuples, smart addressing is not used.

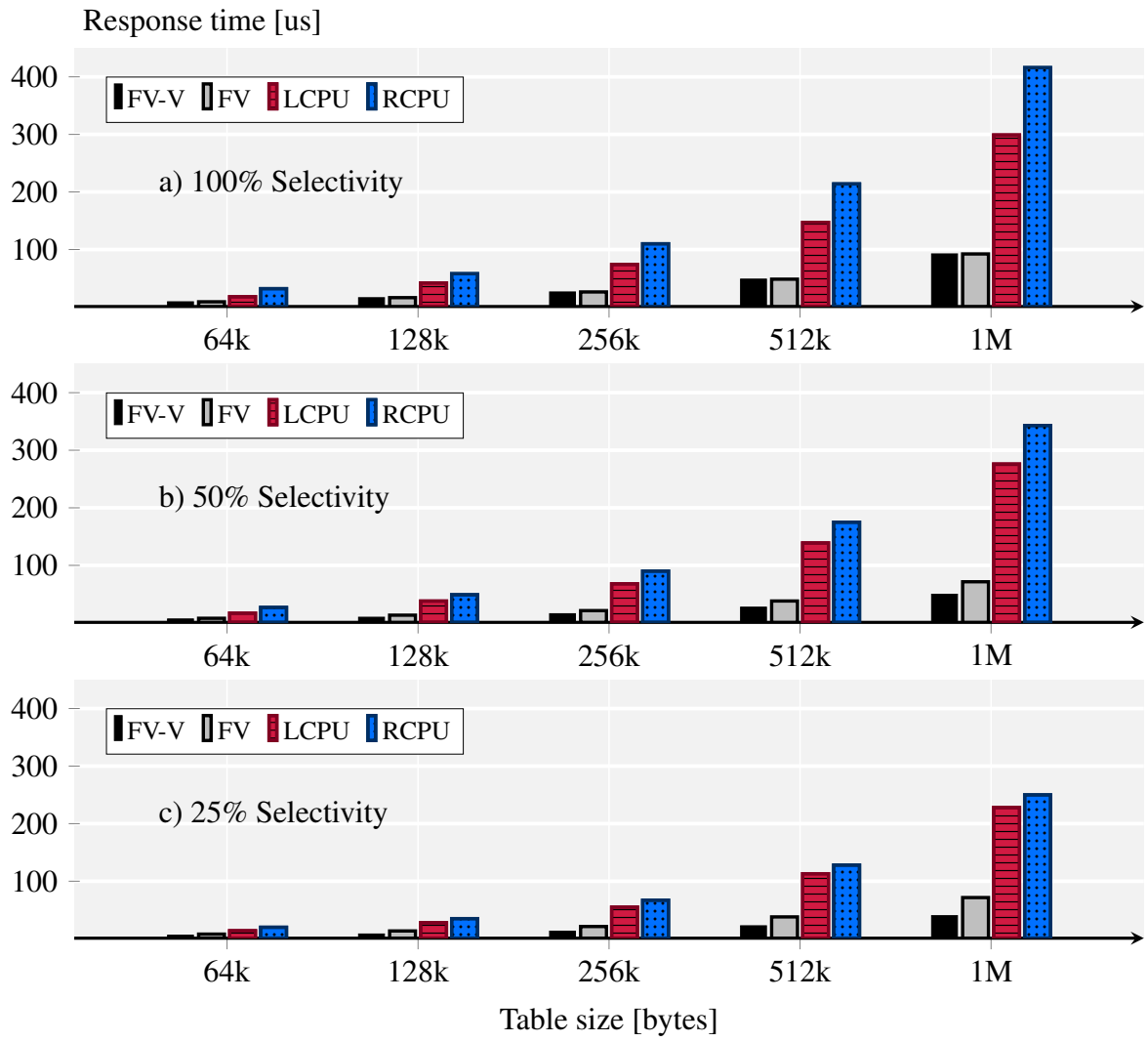

Figure 6.9: Response times for selection queries with 100%, 50% and 25% selectivity, respectively.

6.6.3 Selection

As mentioned, selection in Farview can have more than one predicate, even on multiple columns. The overhead of the selection on the overall processing is negligible and is fully hidden behind memory operations. For the evaluation, we run the following query:

```
SELECT * FROM S WHERE S.a < X AND S.b < Y;
```

We compare Farview with the aforementioned baseline systems. The 64-byte tuples are equal to the width of the pipeline, which leads to the highest pipeline throughput (i.e., a tuple is forwarded each cycle). We vary the selectivity of the query and present our results in Figures 6.9.

As we observe, in all cases (**FV, FV-V**) Farview outperforms both **LCPU** and **RCPU**. **LCPU** incurs high overheads, as it must read the input data from DRAM (not from the processor cache) and then write the results back to DRAM. The **RCPU** baseline additionally has to transfer the data through the network, leading to consistently slower response times than **LCPU**. The advantage of the bump-in-the-wire processing present in Farview is clear, especially as the amount of data becomes larger. We now go into specific details for each selectivity level.

On Figure 6.9(a), the query does not discard any tuples and it fetches the whole table. The query is equivalent to executing:

```
SELECT * FROM S;
```

Since no data is excluded from the selection predicate, **FV** and **RCPU** send the whole table through the network. Farview has similar performance for the vectorized (**FV-V**) and non-vectorized (**FV**) models of processing, because the available network bandwidth is the bottleneck (i.e., parallelization does not provide additional benefit). The memory bandwidth of the remote node is thus underutilized in this specific scenario.

Figure 6.9(b)'s 50% selectivity query alleviates the pressure on the network and permits more utilization of the DRAM bandwidth, leading to higher overall performance for Farview. In this case, the vectorized model is slightly more performant than the standard execution model, as the parallelization of the processing allows the reads from DRAM to occur at higher speeds. Still, even at this selectivity, the network is the bottleneck and the DRAM bandwidth is not fully utilized for a single client. The execution times of the baselines improve relative to 100% selectivity, especially as the input size grows, but they are still slower than both of Farview's execution models.

With 25% selectivity (Figure 6.9(c)), only a small portion of the data is sent over the network, so the network is no longer the bottleneck. For **LCPU**, the data movement between the DRAM and the CPU is reduced because less data is written back; as a result, its performance is better when compared to 50% selectivity, but still worse than Farview. For Farview's non-vectorized model (**FV**), the bottleneck shifts to the bandwidth of a single query pipeline and performance is similar to the 50% selectivity case. The pipeline parallelization of the vectorized model (**FV-V**) can fully utilize the available memory bandwidth, and thus **FV-V** is roughly twice as fast as **FV**.

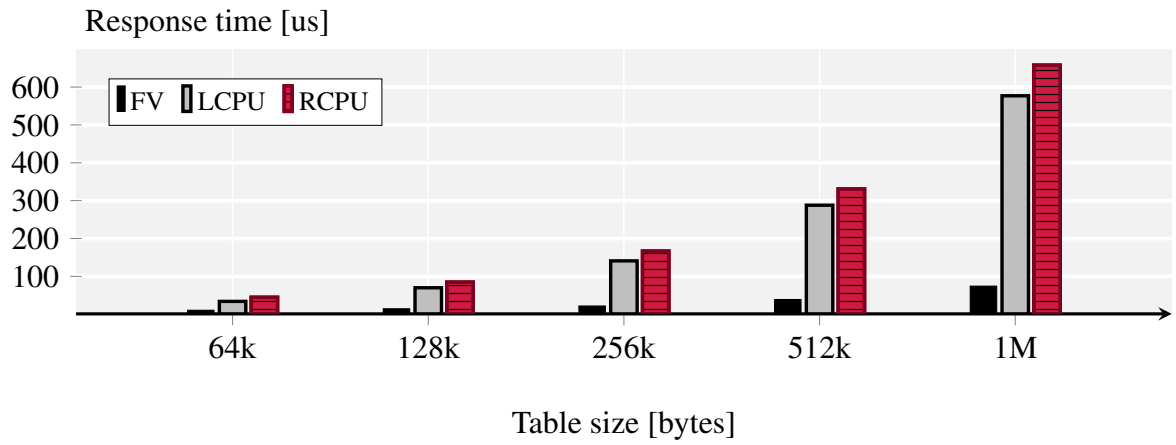

Figure 6.10: Response times for the distinct operator.

6.6.4 Grouping

In this section we evaluate the performance of the `DISTINCT` and `GROUP BY` grouping operators. We use the same baselines as in the selection experiments. For these operators, our baselines use a hashing implementation based on a very fast hash map library[2].

Figure 6.10 presents the results for the following query:

```
SELECT DISTINCT(S.a) FROM S;
```

The number of distinct elements is randomized and the results are averaged out over a number of runs. For the cuckoo hash table implemented in Farview, we assume that no hash collisions occur. In the case of collisions, they would be written in an overflow buffer and sent to the CPU for post-processing. Farview outperforms both baselines, and the baseline runtimes increase dramatically as the input size gets larger. We attribute part of the difference to reading from/writing to DRAM, as in the case of selection. Two additional factors contribute to the slowdown of the baselines: 1) the memory resizing of the hash table as more elements are added and 2) the ability of FPGAs to hash much faster than CPUs, even when the hash function is complex [119]. Additionally, in Farview the query is fully pipelined and there is not much overhead compared to the base selection. Finally, the number of distinct tuples has an impact on performance, similar to the impact of selectivity in the selection experiments. The lower the number of distinct elements is, the lower the amount of data moved through the network, leading to better performance.

[2]https://github.com/greg7mdp/parallel-hashmap

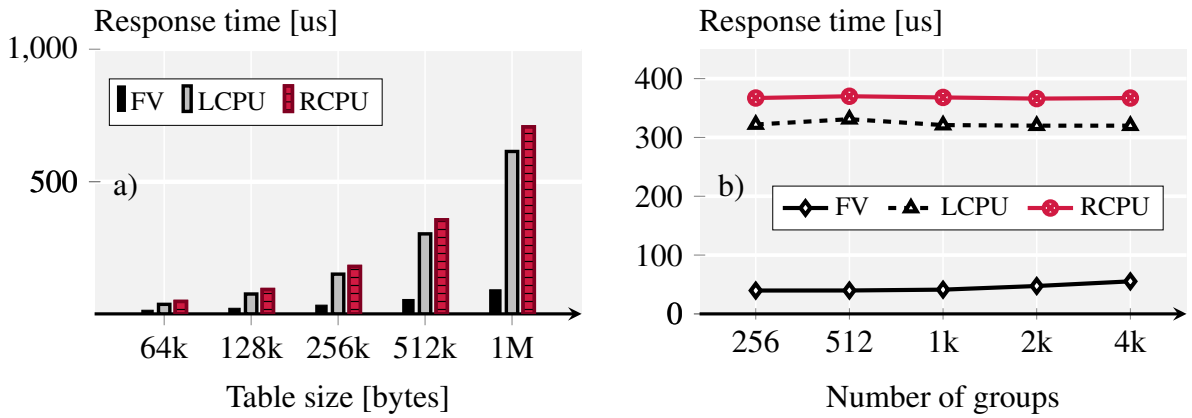

Figure 6.11: Response time comparisons for (a) a group by query with aggregation on an increasing number of elements, and (b) a group by query by with aggregation on a stable number of groups.

Next, we execute a simple query that has a `GROUP BY` and an aggregation (`SUM`):

```
SELECT S.a, SUM(S.b) FROM S GROUP BY S.a;
```

In Figure 6.11(a) and Figure 6.11(b), we present the performance of the query for increasing data sizes and number of tuples, respectively. The `GROUP BY` operator performs a similar operation as the `DISTINCT` operator. The main difference is that for `GROUP BY`, we calculate aggregated statistics for each of the distinct tuples in the hash table. After the aggregation is complete, all of the (unique) entries in the hash table along with their aggregations are sent over the network. This process adds a small amount of latency to the operator execution. The response time is thus bigger if the number of aggregates is higher. Even with this added latency, Farview outperforms the **LCPU** and **RCPU** baselines for both experiments, for the same reasons that were mentioned in the distinct experiment.

6.6.5 Regular Expression Matching

We compare the performance of Farview and the baselines for regular expression matching for different string sizes, where the regular expression matches 50% of the generated strings (Figure 6.12. The baselines use the highly optimized Google *RE2* regular expression library [96]. **FV** outperforms both **LCPU** and **RCPU**. **FV's** regular expression operator is able to sustain the full line rate regardless of the predicate complexity, due to its use of deep pipelining and parallel regular expression engines, which take advantage of the spatial architecture of the FPGA. This

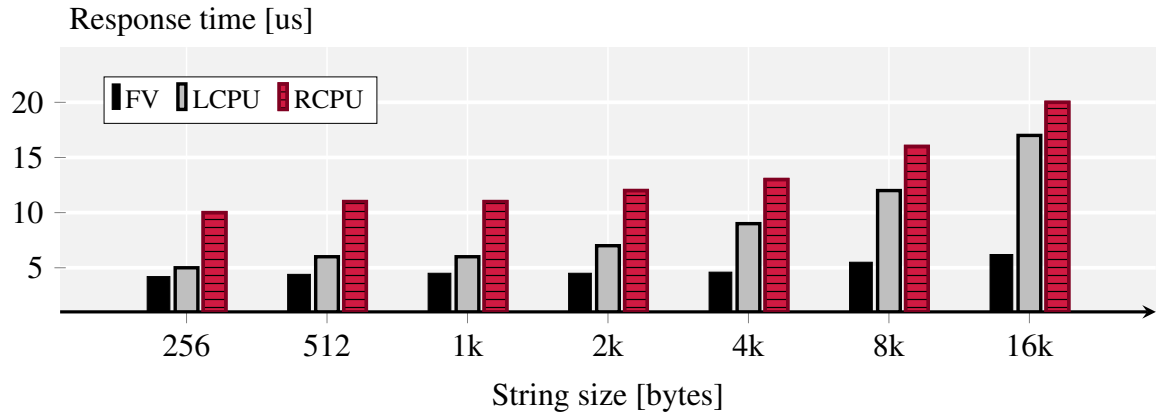

Figure 6.12: Regular expression matching.

implementation outperforms *RE2*, as the implementation on the CPU has far less parallelization potential and the overhead of the data movement from/to DRAM is quite high.

6.6.6 Encryption/Decryption and Multiple Clients

As an example of a commonly used system support operator, we explore encryption/decryption. The encryption algorithm used in Farview is a 128-bit AES in parallelized counter mode. The operator can be placed at the beginning or the end of the operator pipelines (handling decryption and encryption) with only a small extra overhead. This allows Farview to decrypt data residing in disaggregated memory for processing and sending it to the client; to encrypt data after processing to secure the transmission to the client; or to decrypt the data, process it, and encrypt it again for transmission. The response time graph in Figure 6.13(a) compares the time taken to decrypt data being read in Farview and in the two baselines. The ability of the FPGA to sustain line rate processing yields a big advantage, as the overhead of the encryption is fully hidden. **FV** significantly outperforms the **LCPU** and **RCPU** baselines, which use the same encryption/decryption scheme through the *Cryptopp* [72] library. The difference in execution time is caused by both the *cold* caches as well as the decryption overhead in the CPU compared with the highly parallelizable AES implementation available in the FPGA.

The throughput graph in Figure 6.13(b) compares an RDMA read operation in Farview (**FV-RD**) and the same operation together with the decryption (**FV-RD+Dec**) on the read data stream. As the throughput graph shows, there is no noticeable performance penalty, indicating that encryption/decryption can be easily combined with all the previous operators without changing the overall performance.

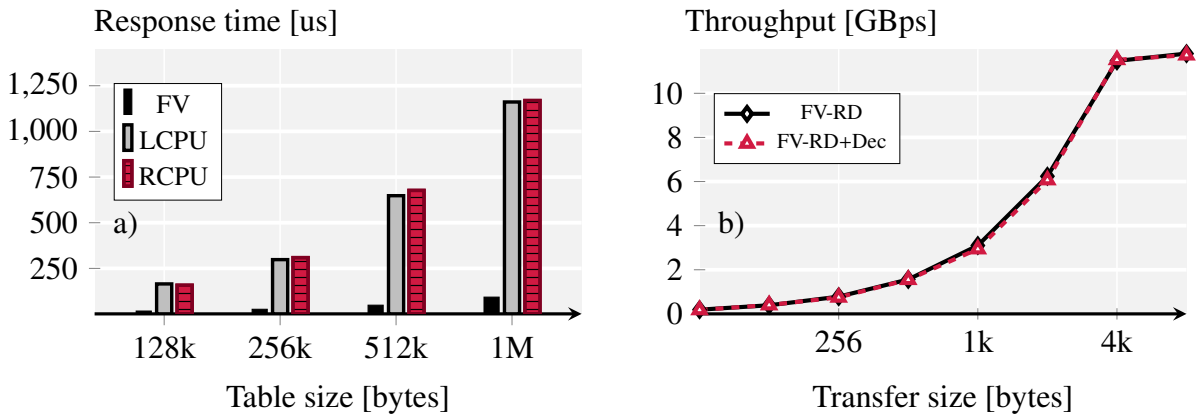

Figure 6.13: Encryption response time (a) and throughput (b).

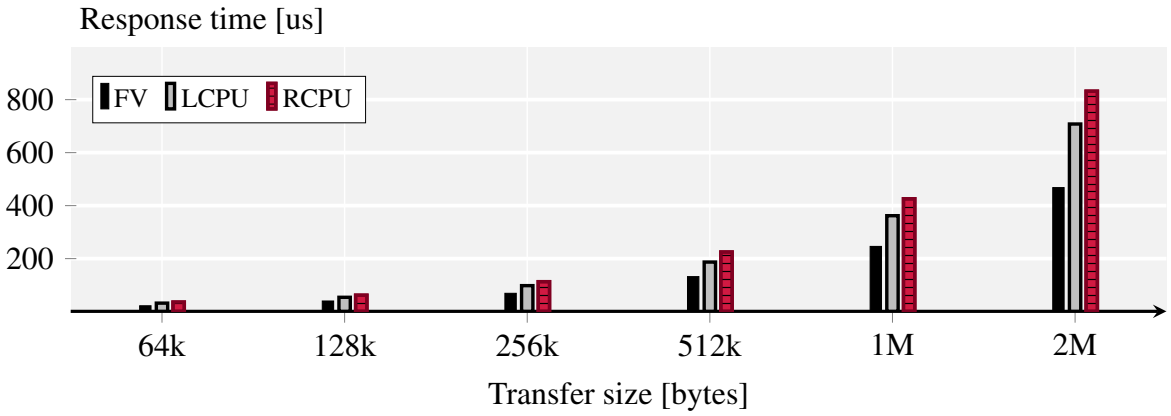

Figure 6.14: Multiple clients.

We finally evaluate the behaviour of the system with multiple clients reading from memory at the same time. We use six clients running the *distinct* query in both Farview and the two CPU baselines (**LCPU**, **RCPU**). The number of distinct elements is small to prevent the network from becoming the main bottleneck and to maximize DRAM performance in all of the clients. For the CPU baselines, we use MPI with 6 processes.

The measurements shown in Figure 6.14 represent the time taken until all six client queries have completed. Farview achieves better performance than both CPU baselines due to the spatial parallelization between multiple dynamic regions, each containing a separate client. The decoupling of the DRAM and the fair sharing provide optimal distribution of the DRAM bandwidth between all dynamic regions and their clients. Both CPU baselines compete for access both to the DRAM and the shared caches, causing interference that affects the overall performance.

6.7 Farview Frontend

Farview is tailored for database workloads, but the programmatic interface we discussed in Section 6.4.2 is implemented in C++ and offers a level of interaction with the system that is much lower than what database clients typically prefer. Ideally, these clients would like to write standard SQL queries to interact with the system.

To bridge this gap, a frontend database query engine is needed. The objective is for this engine to accept standard SQL queries as input, while on the other end, it should contain the interface to communicate with the programmatic interface in Farview.

Therefore, there is a requirement for a versatile and adaptable database query engine that aligns with our specific requirements. Modularis [127] is an open-source data analytics execution layer that fits these needs well for the following reasons:

- Modularis's primary contribution lies in its adaptability to diverse hardware backends with minimal alterations to the rest of the framework. This feature aligns with Farview's requirements, as it enables us to concentrate only on the interface between Farview and Modularis, without the need to address the higher layers of the engine's stack.
- Modularis offers compatibility with existing RDMA backends, as it includes integration with RDMA libraries. It thus already contains the similar interfaces as ones needed for the integration with Farview.
- Modularis is built on the concept that database operators adhere to common interfaces. As a result, it divides larger operators into smaller sub-operators with clearly defined interfaces. This approach enables high levels of modularity and encourages reuse. This modularity is what ultimately enables us to integrate Farview.

The concept is to employ Modularis as the client-side execution engine, which will subsequently communicate with Farview's disaggregated nodes through RDMA. The primary advantage of having Modularis in the frontend is the ability to push down specific operators, such as projections and selections. This approach significantly enhances overall performance, as illustrated by Farview already. This approach enables the frontend database engine to implement a variety of optimizations and allows end users to compose queries in traditional high-level languages without the need for direct interaction with Farview.

It's important to note that our objective is not to offload complete queries to Farview nodes. Rather, by leveraging Modularis, we gain the capability to effectively identify the sub-operators, typically located in the lower sections of the query plan, that are suitable for offloading to Farview (selections, projections, etc.).

6.7.1 Modularis

The execution of queries in Modularis begins with clients composing queries within the Python frontend, where they also specify the target platform. This is where we introduce the additional option for Farview as a backend.

The parser then analyzes the input query to generate a Directed Acyclic Graph (DAG). This DAG is subsequently serialized and handed over to the C++ backend, where the bulk of transformations and optimizations take place. Once these are completed, the outcome is a pipeline comprising sub-operators that communicate through standardized interfaces (Volcano model [98]). These interfaces encompass simple functions that operate on individual tuples, including open, close, and next calls. Provided that the sub-operators adhere to these interfaces, they can be customized to meet specific requirements and, therefore, be adapted to various hardware platforms.

We integrate Farview into Modularis by introducing Farview-specific hardware sub-operators. This way, Farview transparently becomes an extension backend for Modularis.

6.7.2 Integration

We will now explain the high-level overview of the integration of Farview into Modularis. Detailed implementation specifics are beyond the scope of this thesis and can be explored in [216].

Figure 6.15 shows the complete system with Farview in the backend and Modularis in the frontend. Modularis serves as the database engine and operates on the client machines, while Farview is deployed on the disaggregated server nodes. The essential high-performance network communication is facilitated through RDMA. In this setup, Modularis sends queries to Farview, which then processes them and sends back the results to the clients.

On Farview side, we introduce a new hardware query parser tasked with interpreting the Modularis queries submitted via the two-sided `SEND` verb. This parser operates by extracting information from the receive queues. It extracts essential information such as the columns to project, predicates, table locations, transfer sizes, and more. Utilizing this information, the parser trig-

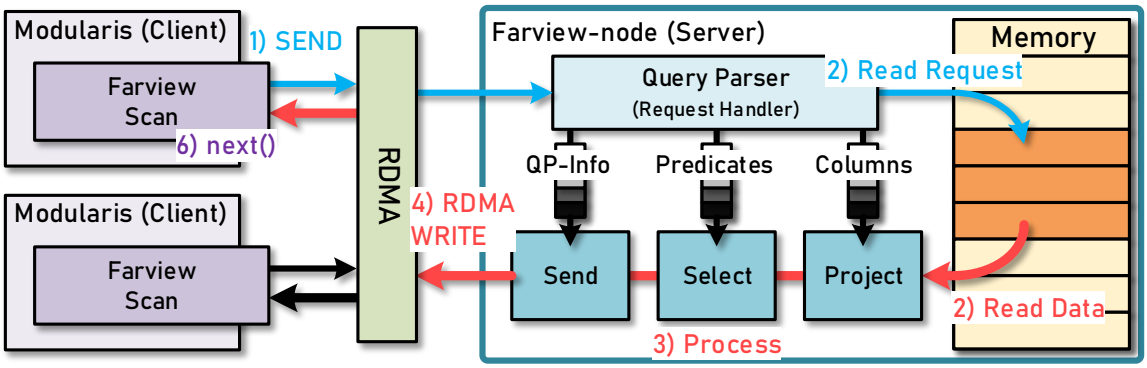

Figure 6.15: Modularis and Farview integration.

gers a request to Farview's memory stack to retrieve the requested table and its tuples. Simultaneously, it initializes the operator pipeline to process this query. These operator pipelines are flexible sequences of operators that we've already discussed in the context of Farview.

The primary modification in Modularis involves the introduction of a new sub-operator, the *Farview Scan*. This operator establishes a network connection with Farview nodes (through queue pairs) and issues query requests, offloading common projection and selection operators. The sub-operator completely aligns with the Volcano operating model within Modularis, making the integration a straightforward process. In this model, each database operator processes individual tuples sequentially, and different operators can work on processing tuples concurrently. This aligns well with the existing computational model in Farview, where tuples are also processed sequentially within internal hardware pipelines.

For our initial testing of the Farview and Modularis integration, we've created an operator pipeline that combines both projection and selection operations to create the hardware *Farview Scan* operator.

Table 6.2: Farview resource usage with the *Farview Scan* operator.

Configuration	CLB LUTs	Regs	BRAM tiles	DSPs
Farview Scan (+ the whole shell)	18.2%	17.3%	36.2%	0%

6.7.3 Initial Evaluation of a Compete System

Experiments are executed, yet again, on the HACC cluster at ETH. For these tests we use the Alveo-u55c [15] cards, equipped with 16GB of HBM memory. The baselines are running on the 16-core AMD EPYC CPUs. The processors are equipped with 1KiB of L1 cache, 8MiB of L2 per core, as well as 128MiB total L3 cache. The clock speed of the CPU is 3GHz. The resource usage of the bitstream with projection and selection pipeline is shown in Table 6.2. In this case, the largest usage is the on-chip memory which has been used generously for queues to decouple the operators within the pipeline.

In the baseline test, Modularis is executed on a single server using a single-core configuration, and it exclusively employs pre-existing sub-operators. Before each hardware test with Farview, once the connection is initially set up, the table is loaded into the disaggregated memory node. Following this initial loading, the table data is treated as read-only.

To compare Modularis with and without Farview and show the advantages of reducing data transfer, a series of tests involving projections and selections are conducted. The SQL code used to execute these queries is as follows:

```
SELECT S.a, S.b, S.e
FROM ex_relation S
WHERE S.a < p1 OR S.b < p2 OR S.e < p3 OR S.h >= p4;
```

The test varies the selectivity within the queries (parameters `p1`, `p2`, `p3` `p4`), similar to what was done in base Farview tests of the selection operator. The table used has a size of 2MB and consists of 32k rows. Each test is averaged over 100 runs.

The results obtained are presented in Figure 6.16. They show a comparison of throughput (measured in queries per second) between running a query entirely within Modularis and offloading the query to Farview, accross a range of selectivity values.

We can infer from the results that Farview outperforms the baseline Modularis for all selectivity values. At lower selectivity levels, Farview can efficiently filter the data with minimal cost. As a substantial portion of the data is filtered out, the CPU only needs to process a small number of tuples on the client side, thus the added data movement is minimal. In contrast, at higher selectivity levels, Farview doesn't filter out as much data, and consequently, the CPU has to iterate over most of the tuples. In such cases, the advantage of offloading selection and projection operations decreases. While these are just preliminary tests, they already indicate that offloading

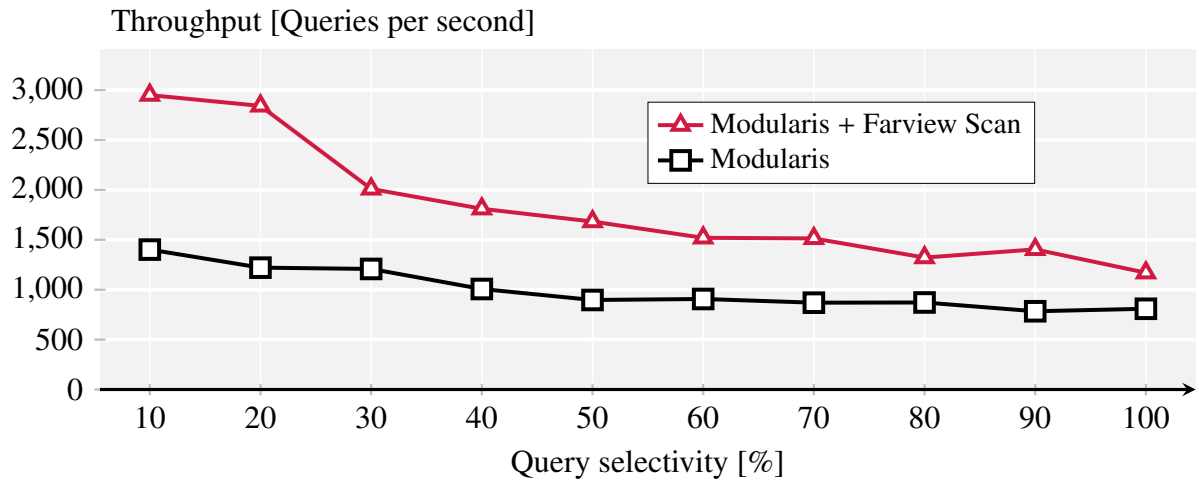

Figure 6.16: Comparison between base Modularis and with added *Farview Scan*.

these basic database operations to a smart disaggregated memory solution like Farview can yield performance benefits. More comprehensive testing on this platform is reserved for future work.

6.8 Summary

Farview implements network-attached disaggregated memory with the capability to offload query processing operators directly to the memory. In this chapter, we have discussed the design of Farview and how it helps to address DRAM capacity challenges (by allowing us to move the buffer pool to a central location) and data movement inefficiencies (by enabling near-data processing to filter the data before it is sent through the network).

Through the use of RDMA, Farview provides performance that is comparable to that attainable using local memory, a performance advantage that is augmented by the ability to process data in-situ. The next steps for the Farview project are to further explore the integration into Modularis, to design suitable cache management strategies to move data back and forth to persistent storage, and to expand the range of operators supported, which might open up the potential for offloading larger portions of the queries.

Chapter 7

Conclusion

Modern computing hardware is constantly evolving, and it is a far cry from traditional load/store architectures. The emergence of multicore processors, GPUs, specialized accelerators, high-speed interconnects and highly flexible FPGAs has given rise to a diverse and heterogeneous computing landscape, introducing in a considerable level of complexity.

Regrettably, the system software abstractions for this innovative hardware are lagging behind, predominantly maintaining a simplistic and outdated view on modern hardware. Consequently, conventional set of abstractions in current use proves insufficient when applied to these diverse novel systems and must be extended to encompass an entire range of new functionalities.

Through this thesis, our goal was to overcome these limitations by examining how traditional system software abstractions can be adapted and expanded to exploit the vast potential of modern heterogeneous systems. Our primary focus was on hybrid computing systems that combine general-purpose CPUs with FPGAs, primarily because these devices offer a vast array of computational possibilities and heterogeneous configurations due to their remarkable adaptability. These hybrid systems allowed us to investigate diverse memory structures and leverage the extensive networking capabilities of these devices to explore distributed systems that extend beyond the limits of a single node.

The outcome of this thesis is Coyote, a comprehensive ”hardware operating system” designed for modern heterogeneous platforms incorporating FPGAs. Coyote, akin to a microkernel, offers a foundational set of essential abstractions upon which additional services can be built. By doing so, it serves as the potential foundation for additional research into these novel architectures.

In *Chapter 2*, we offered an in-depth examination of the existing abstractions within modern heterogeneous systems. We illustrated how traditional operating system abstractions can be applied to these hybrid computing platforms. We explored familiar abstractions and services, including processes and context switching, virtual and physical memory management, scheduling, networking and IPC. Additionally, we set the initial framework for the development of Coyote.

Chapter 3 dived into the implementation details of Coyote. At this stage, we explored the lowest levels of the system, focusing on the organization of the static layer, which, in some aspects, can be considered as the core or kernel of the entire system. We showed how this static layer is structured and how it interacts with the host CPU hardware and software. This layer ensures system-wide portability, provides a uniform execution environment, and forms the foundation for a multi-tenant environment in our system. We then gave a practical demonstration of these principles by successfully porting Coyote to Enzian, a custom-designed research computer. Additionally, we investigated the hierarchical reconfiguration, which introduces an innovative approach to improving system flexibility.

Chapter 4 took us a step further, concentrating on one of the most important abstractions in operating systems, the virtual memory. We showed numerous significant advantages this abstraction can bring when applied to FPGAs, and investigated the hardware and software structures that enable its implementation. In the latter part of this chapter, our focus shifted to what is arguably the most powerful abstraction in modern FPGAs: the network services. We integrated both the TCP stack, in use for majority of the modern networking, and a high performance RDMA stack.

In *Chapter 5*, we arrived at the application layer. This chapter primarily focused on the user-facing interfaces provided by the system. We demonstrated how our earlier abstractions enable us to unify interfaces, both in software and hardware, significantly simplifying interactions with the system as a whole (which was one of the main goals). Additionally, we explored how Coyote can facilitate the interaction between hardware tenants and virtual machines, offering a model for sharing multi-tenant FPGAs in the cloud. Lastly, we showcased how practical applications can fully leverage Coyote's capabilities and how additional services can be developed on top.

Having completed our exploration of the foundational Coyote system, *Chapter 6* explored how our "microkernel" can be employed to investigate common computing paradigms, like disaggregation. We introduced Farview, a system constructed on top of Coyote, which offers smart disaggregated memory to address the substantial data movement bottlenecks and memory capacity limitations encountered in modern databases. It achieves this by bringing computation closer to the data, by offloading traditional database operators and performing "on-the-fly" computations.

Through the systems developed throughout this thesis we have demonstrated a set of contemporary hardware and software abstractions for modern heterogeneous hardware. We hope that these open-source end-to-end systems can play a role in bridging the gap between modern hardware and system software development, while also making meaningful contributions to future research.

List of Figures

List of Tables

List of Listings

BIBLIOGRAPHY

[1] 2002. *POSIX Threads and the Linux Kernel*. IBM. Accessed: 2023-10-30.

[2] 2005. *Secure Virtual Machine Architecture Reference Manual*. AMD. Accessed: 2023-10-30.

[3] 2016. *Vivado Design Suite User Guide - Partial Reconfiguration; Xilinx User Guide 909*. Xilinx.

[4] 2018. *Intel Scalable I/O Virtualization*. Intel. Accessed: 2023-10-30.

[5] 2019. *ADM-PCIE-7V3 Datasheet*. Alpha Data.

[6] 2019. *Intel FPGA Programmable Acceleration Card D5005 Data Sheet*, ds-1058 edition. Intel. https://www.intel.com/content/www/us/en/programmable/documentation/cvl1520030638800.html.

[7] 2019. *SDAccel Environment User Guide*, v2019.1 edition. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf.

[8] 2020. *Alveo U200 Data Center Accelerator Cards Data Sheet*, v.1.3.1 edition. AMD. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[9] 2020. *Alveo U250 Data Center Accelerator Cards Data Sheet*, v.1.3.1 edition. AMD. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[10] 2020. *Alveo U280 Data Center Accelerator Card Data Sheet*, v.1.3 edition. AMD. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[11] 2020. *Data Plane Development Kit - Programmer's Guide*. Intel.

[12] 2021. *Alveo data center cards*, v.1.9 edition. AMD. https://docs.xilinx.com/v/u/1.9-English/ug1301-getting-started-guide-alveo-accelerator-cards.

[13] 2022. *ICAP Interface*, v.4.1 edition. AMD. https://docs.xilinx.com/r/en-US/pg036_sem/ICAP-Interface.

[14] 2023. *Alveo u50 Data Center Accelerator Cards Data Sheet*, v.1.8 edition. AMD. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html.

[15] 2023. *Alveo u55c Data Center Accelerator Cards Data Sheet*, v.1.3 edition. AMD. https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html.

[16] 2023. *CUDA Toolkit Documentation*. NVIDIA.

[17] 2023. *GPUDirect RDMA*. NVIDIA.

[18] 2023. *Heterogeneous Memory Management (HMM)*. Linux.

[19] 2023. *Intel UPI*. Intel.

[20] 2023. *VCU 118 Xilinx Development Board*, v.1.5 edition. Xilinx. https://www.xilinx.com/products/boards-and-kits/vcu118.html.

[21] 2023. *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*, 2023.1 edition. AMD. https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration.

[22] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, pages 775–787.

[23] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 121–127.

[24] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 120–126.

[25] Kathirgamar Aingaran, Sumti Jairath, and David Lutz. 2016. Software in silicon in the oracle SPARC M7 processor. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–31.

[26] Alibaba Cloud Services. 2020. Compute optimized instance families with FPGAs. https://www.alibabacloud.com/help/doc-detail/108504.htm.

[27] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: the data processing interface for modern networks. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

[28] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Owaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands.

[29] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 14:1–14:16.

[30] Amazon Web Services. 2020. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[31] AMD. 2016. AXI HWICAP v3.0 LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf.

[32] AMD. 2019. Versal AI Core. https://old.hotchips.org/hc31/HC31_2.5_Xilinx_Versal_Hotchips31_Final_v2.pdf.

[33] AMD. 2023. UltraScale+ Devices Integrated Block for PCI Express. https://docs.xilinx.com/viewer/book-attachment/tCJDwogjyJ9_CE2Q_px23A/B6WE5d4mRhWPAlUpzHxkcw.

[34] AMD Xilinx. 2019. DMA/Bridge Subsystem for PCI Express v4.1 Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[35] AMD Xilinx. 2022. QDMA Subsystem for PCI Express. https://docs.xilinx.com/viewer/book-attachment/n6fBd_xlVtlFE96gGACfJA/gbwHDtOsU98Gh7U6BMS5rg.

[36] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter J. Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28.

[37] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 96–107, New York, NY, USA. Association for Computing Machinery.

[38] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *ICDE'15*, pages 435–446.

[39] Andrea Arcangeli. 2008. Integrating KVM with the Linux Memory Management. https://www.linux-kvm.org/images/3/33/KvmForum2008%24kdf2008_15.pdf. Accessed: 2023-10-15.

[40] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758.

[41] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile memory database management systems. *Synthesis Lectures on Data Management*, 11.

[42] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*.

[43] S. Asano, T. Maruyama, and Y. Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131.

[44] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. 2017. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access*, 5:1900–1910.

[45] AWSCloud. 2020. Aqua (advanced query accelerator) for amazon redshift.

[46] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *ACM Queue*, 11:40:40–40:52.

[47] Donald G. Bailey. 2015. The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, pages 134–139. ACM.

[48] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1463–1475.

[49] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5):517–528.

[50] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. *Proc. VLDB Endow.*, 12(13):2325–2338.

[51] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A `fork()` in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, Bertinoro, Italy.

[52] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA. USENIX Association.

[53] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19.

[54] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85.

[55] K. M. Bresniker, S. Singhal, and R. S. Williams. 2015. Adapting to thrive in a new economy of memory abundance. *Computer*, 48(12):44–53.

[56] Nick Brown. 2020. Exploring the acceleration of nekbone on reconfigurable architectures. *CoRR*, abs/2011.04981.

[57] William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance.

[58] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing Virtualization to the X86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4).

[59] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE.

[60] Anthony M. Cabrera and Roger D. Chamberlain. 2019. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proceedings of the International Workshop on OpenCL*.

[61] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. ASPLOS 2021, page 79–92.

[62] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[63] CCIX Consortium and others. 2019. Cache Coherent Interconnect for Accelerators (CCIX). `http://www.ccixconsortium.com`.

[64] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 3. ACM.

[65] Steven Wei Der Chien, Ivy Bo Peng, and Stefano Markidis. 2019. Performance evaluation of advanced features in CUDA unified memory. *CoRR*, abs/1910.09598.

[66] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in sap hana. *Proc. VLDB Endow.*, page 3277–3291.

[67] Monica Chiosa, Thomas B. Preußer, and Gustavo Alonso. 2021. Skt: A one-pass multi-sketch data analytics accelerator. 14(11):2369–2382.

[68] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, pages 8–20.

[69] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106. ACM.

[70] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 434–451. Association for Computing Machinery.

[71] Oracle Corporation. 2017. Accelerating Spark SQL Using SPARC with DAX.

[72] CryptoPP. Accessed 2021. Crypto++® library 8.6. `https://www.cryptopp.com/`.

[73] CXL Consortium. 2020. Compute Express Link. `https://www.computeexpresslink.org/`.

[74] Paul Dahlke. 2023-09-12. Hypervisor for a multi-tenant fpga heterogeneous system. Master thesis, ETH Zurich.

[75] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. 2005. Design and implementation of the iwarp protocol in software. pages 471–476.

[76] William Dally. 2011. Power, programmability, and granularity: The challenges of exascale computing. In *2011 IEEE International Test Conference*, pages 12–12. IEEE Computer Society.

[77] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57.

[78] Norbert Deak, Octavian Cret, and Horia Hedesiu. 2019. Efficient fpga floorplanning for partial reconfiguration-based applications. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 309–309.

[79] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA. USENIX Association.

[80] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Dylan Rankin, Ryan Rivera, Sioni Summers, Nhan Tran, and Zhenbin Wu. 2019. Fast inference of deep neural networks for real-time particle physics applications. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 305. Association for Computing Machinery.

[81] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *European Conference on Computer Systems*.

[82] Lieven Eeckhout. 2017. Is moore's law slowing down? what's next? *IEEE Micro*, 37(4):4–5.

[83] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in facebook. In *European Conference on Computer Systems (EuroSys) 2018*.

[84] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134.

[85] J Evans and S Georgakakis. 2011. Overview of high level synthesis tools. *JINST*.

[86] Suhaib Fahmi, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE.

[87] Yuanwei Fang, Chen Zou, and Andrew Chien. 2019. Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture. In *Proc. VLDB Endow.*

[88] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[89] Yan Feng and D.P. Mehta. 2006. Heterogeneous floorplanning for fpgas. In *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, pages 6 pp.–.

[90] Johannes de Fine Licht and Torsten Hoefler. 2019. hlslib: Software engineering for hardware design. *arXiv:1910.04436*.

[91] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, 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.

[92] D. Fortún, C. G. de la Cueva, J. Grajal, M. López-Vallejo, and C. L. Barrio. 2018. Performance-oriented Implementation of Hilbert Filters on FPGAs. In *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6.

[93] Carsten Binnig Andrew Crotty Alex Galakatos and Tim Kraska Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7).

[94] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *OSDI*, pages 249–264.

[95] Alex Goldhammer and John Ayer. 2008. Understanding performance of pci express systems.

[96] Google. Accessed 2021. re2 library. `https://github.com/google/re2`.

[97] B. Gottschall, T. Preußer, and A. Kumar. 2018. Reloc – An Open-Source Vivado Workflow for Generating Relocatable End-User Configuration Tiles. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–211.

[98] G. Graefe. 1994. Volcano— an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, page 120–135.

[99] Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record*, 19(2):102–111.

[100] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. 2006. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9.

[101] ETH Zurich Systems Group. 2020. Enzian, a research computer. `http://enzian.systems`.

[102] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667.

[103] Zhenhao He. 2020. Bit-Serial kmeans. `https://github.com/fpgasystems/bit_serial_kmeans`.

[104] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203.

[105] Zhenhao He, Daniele Parravicini, Lucian Petrica, Kenneth O'Brien, Gustavo Alonso, and Michaela Blott. 2021. Accl: Fpga-accelerated collectives over 100 gbps tcp-ip. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 33–43.

[106] Zhenhao He, Zeke Wang, and Gustavo Alonso. 2020. Bis-km: Enabling any-precision k-means on fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 233–243. Association for Computing Machinery.

[107] InfiniBand. Accessed 2021. Infiniband™ Architecture Volume 1 and Volume 2. `https://www.infinibandta.org/ibta-specification/`.

[108] InfiniBand. Accessed 2021. Infiniband™architecture specification release 1.2.1 annex a16: Roce. https://cw.infinibandta.org/document/dl/7148.

[109] Intel. Intel high level synthesis compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. Accessed: 2022-09-21.

[110] Intel Corporation. 2009. An Introduction to the Intel QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[111] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, page 1202–1213.

[112] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA. USENIX Association.

[113] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. Yoursql: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935.

[114] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

[115] C. Kachris and D. Soudris. 2016. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10.

[116] Torben Kalkhof and Andreas Koch. 2021. Efficient physical page migrations in shared virtual memory reconfigurable computing systems. In *2021 International Conference on Field-Programmable Technology (ICFPT)*.

[117] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer.

In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA. Association for Computing Machinery.

[118] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. 2017. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167.

[119] K. Kara and G. Alonso. 2016. Fast and robust hashing for database operators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4.

[120] Kaan Kara and Gustavo Alonso. 2020. Pipearch: Generic and context-switch capable data processing on fpgas.

[121] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52.

[122] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127.

[123] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. HyperLoop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297—-312.

[124] Vipin Kizheppatt and Suhaib Fahmy. 2014. ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq. *Embedded Systems Letters, IEEE*, 6:41–44.

[125] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. 2017. Migration of long-running Tasks between Reconfigurable Resources using Virtualization. *ACM SIGARCH Computer Architecture News*, 44(4):56–61.

[126] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated memory with operator off-

loading for database engines. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*.

[127] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: modular relational analytics over heterogeneous distributed platforms. *Proc. VLDB Endow.*, page 3308–3321.

[128] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. 2020. HyperLogLog Sketch Acceleration on FPGA. *arXiv e-prints*, page arXiv:2005.13332.

[129] I. Kuon and J. Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215.

[130] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, page 705–721. USENIX Association.

[131] H. Andrés Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330.

[132] Nikita Lazarev, Shaojie Xiang, neil Adit, Zhiru Zahng, and Christina Delimitrou. 2021. Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *ASPLOS*.

[133] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. Kv-direct: High-performance in-memory key-value store with programmable nic. SOSP '17, page 137–152. Association for Computing Machinery.

[134] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, page 2–es. Association for Computing Machinery.

[135] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587.

[136] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098.

[137] X. Li, L. Ding, L. Wang, and F. Cao. 2017. FPGA accelerates deep residual learning for image recognition. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 837–840.

[138] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. 2012. High-level synthesis: Productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*.

[139] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009)*, pages 267–278.

[140] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 189–200.

[141] Linux. . Heterogeneous Memory Management (HMM). `https://www.kernel.org/doc/html/v5.0/vm/hmm.html`. Accessed: 2023-10-15.

[142] Linux. . Virtual function i/o. `https://docs.kernel.org/driver-api/vfio.html`. Accessed: 2023-10-20.

[143] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Trans. Database Syst.*, 44(4):17:1–17:45.

[144] J. Liu, A. Dragojevic, S. Fleming, A. Katsarakis, D. Korolija, I. Zablotchi, H. Ng, A. Kalia, and M. Castro. 2024. Honeycomb: Ordered key-value store acceleration on an fpga-based smartnic. *IEEE Transactions on Computers*, pages 857–871.

[145] ARM Ltd. 2010. AMBA 4 AXI4-Stream Protocol. `https://developer.arm.com/docs/ihi0051/latest`.

[146] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 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*, ASPLOS '20, page 827–844, New York, NY, USA. Association for Computing Machinery.

[147] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS'20*, pages 757–773.

[148] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. 1991. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25:110–121.

[149] Dimitrios Meidanis, Konstantinos Georgopoulos, and Ioannis Papaefstathiou. 2011. Fpga power consumption measurements and estimations under different implementation parameters. In *2011 International Conference on Field-Programmable Technology*, pages 1–6.

[150] Mellanox. 2020. Mellanox Innova™-2 FlexOpen programmable SmartNIC. `https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf`.

[151] Mellanox Technologies. 2017. RoCE vs. iWARP Competitive Analysis. `https://network.nvidia.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf/`.

[152] Onur Mutlu. 2019. Processing data where it makes sense in modern computing systems: Enabling in-memory computation. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, page 5–6. Association for Computing Machinery.

[153] Alexey Natekin and Alois Knoll. 2013. Gradient Boosting Machines, A Tutorial. *Frontiers in neurorobotics*, page 21.

[154] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty, and Mladen Berekovic. 2023. Fpga-based deep learning inference accelerators: Where are we standing? *ACM Trans. Reconfigurable Technol. Syst.*

[155] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference ATC 2015*.

[156] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550.

[157] Thomas Neumann and Michael J Freitag. 2020. Umbra: A disk-based system with in-memory performance. In *CIDR*.

[158] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 677–689, New York, NY, USA. Association for Computing Machinery.

[159] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70.

[160] Eriko Nurvitadhi, Jeffrey J. Cook, Asit K. Mishra, Debbie Marr, Kevin Nealis, Philip Colangelo, Andrew C. Ling, Davor Capalija, Utku Aydonat, Aravind Dasu, and Sergey Y. Shumarayev. 2018. In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 106–110.

[161] NVidia. 2017. NVidia Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[162] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. 2011. A reconfigurable computing system based on a cache-coherent fabric. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 80–85. IEEE.

[163] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA 1998, Barcelona, Spain, June 27 - July 1, 1998*, pages 192–203.

[164] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The ramcloud storage system. *ACM Trans. Comput. Syst.*

[165] Muhsen Owaida and Gustavo Alonso. 2020. Distributed Inference over Decision Tree Ensembles. `https://github.com/fpgasystems/Distributed-DecisionTrees`.

[166] Muhsen Owaida, Amit Kulkarni, and Gustavo Alonso. 2019. Distributed Inference over Decision Tree Ensembles on Clusters of FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 12(4).

[167] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218. IEEE.

[168] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules. In *MEMSYS 2019*, pages 288–303.

[169] David A. Patterson, Thomas E. Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos E. Kozyrakis, Randi Thomas, and Katherine A. Yelick. 1997. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44.

[170] David A. Patterson, Krste Asanovic, Aaron B. Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos E. Kozyrakis, David R. Martin, Stylianos Perissakis, Randi Thomas, Noah Treuhaft, and Katherine A. Yelick. 1997. Intelligent RAM (IRAM): the industrial setting, applications and architectures. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 2–7.

[171] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the intel optane byte-addressable NVM. In *MEMSYS '19: Proceedings of the International Symposium on Memory Systems*, pages 304–315.

[172] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24.

[173] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. 2019. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8.

[174] Avi Qumranet, Yaniv Qumranet, Dor Qumranet, Uri Qumranet, and Anthony Liguori. 2007. Kvm: The linux virtual machine monitor. *Proceedings Linux Symposium*, 15.

[175] Abishek Ramdas, David Cock, Timothy Roscoe, and Gustavo Alonso. 2021. The enzian coherent interconnect (eci): opening a coherence protocol to research and applications. page 18. Capra Research Group, Computer Science and Electrical and Computer Engineering Department, Cornell University.

[176] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555.

[177] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*.

[178] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 233–248, New York, NY, USA. Association for Computing Machinery.

[179] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An fpga-based open-source 100 gbe TCP/IP stack. In *FPL 2019*, pages 286–292.

[180] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. 2015. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. `https://arxiv.org/abs/1505.01120v1`.

[181] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87.

[182] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *ACM Symposium on Cloud Computing (SOCC) 2017*.

[183] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. 2015. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43.

[184] D. Sidler, Z. István, and G. Alonso. 2016. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4.

[185] David Sidler. 2019-09. *In-Network Data Processing using FPGAs*. Ph.D. thesis.

[186] David Sidler, Monica Chiosa, Zhenhao He, Mario Ruiz, Kimon Karras, and Lisa Liu. 2020. Scalable Network Stack supporting TCP/IP, RoCEv2, UDP/IP at 10-100Gbit/s. `https://github.com/fpgasystems/fpga-network-stack.git`.

[187] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. Strom: smart remote memory. In *EuroSys'20*, pages 29:1–29:16.

[188] J. Stuecheli, B. Blaner, C.R. Johns, and M.S. Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1).

[189] Mellanox Technologies. Mellanox connectx-5 card. `https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf`. Accessed: 2023-10-20.

[190] Shelby Thomas, Geoffrey M. Voelker, and George Porter. 2018. CacheCloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA. USENIX Association.

[191] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. 2020. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*.

[192] Stephen M. Trimberger. 2015. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331.

[193] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX ATC 2020*.

[194] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer*, 38(5):48–56.

[195] M. Vaidehi and B.Justus Rabi. 2014. Design and analysis of aes-cbc mode for high security applications. pages 499–502.

[196] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE.

[197] Malte Vesper, Dirk Kocha, and Khoa Phama. 2017. PCIeHLS: an OpenCL HLS framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 10–15. IEEE.

[198] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4).

[199] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2018. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. In *IEEE Transactions on Computers (Volume: 68 , Issue: 4 , April 1 2019)*, pages 510–525. IEEE.

[200] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. 2019. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. *CoRR*, abs/1901.04988.

[201] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. 2019. A survey of fpga based deep learning accelerators: Challenges and opportunities.

[202] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119.

[203] Skyler Windh, Xiaoyin Ma, Robert Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid Najjar. 2015. High-Level Language Tools for Reconfigurable Computing. In *Proceedings of the IEEE (Volume: 103 , Issue: 3 , March 2015)*, pages 390 – 408. IEEE.

[204] F. Winterstein and G. Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in opencl tools for fpgas. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 104–111.

[205] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *Proc. VLDB Endow.*, 7(11):963–974.

[206] Ran Wu, Xinmin Guo, Jian Du, and Junbao Li. 2021. Accelerating neural network inference on fpga-based platforms—a survey. *Electronics*.

[207] AMD Xilinx. . Vitis unified software platform. `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html`. Accessed: 2022-09-21.

[208] AMD Xilinx. . Xrt and vitis platform overview. `https://xilinx.github.io/XRT/master/html/platforms.html`. Accessed: 2023-10-06.

[209] Zeping Xue and D. B. Thomas. 2015. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7.

[210] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 845–858, New York, NY, USA. Association for Computing Machinery.

[211] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. 2017. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, page 4035–4043.

[212] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 22. ACM.

[213] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking data management systems for disaggregated data centers. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Online Proceedings*. www.cidrdb.org.

[214] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):1568–1581.

[215] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758.

[216] Mihai Zorca. 2023. Integrating a smart storage engine into a database system. Master thesis, ETH Zurich, Zurich.

[217] Marcin Zukowski and Peter A Boncz. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27.