Do OS abstractions make sense on FPGAs?

Dario Korolija, Timothy Roscoe, and Gustavo Alonso, *ETH Zurich*

https://www.usenix.org/conference/osdi20/presentation/roscoe

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Do OS abstractions make sense on FPGAs?

Dario Korolija, Timothy Roscoe, Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zurich
`{dario.korolija, troscoe, alonso}@inf.ethz.ch`

Abstract

Hybrid computing systems, consisting of a CPU server coupled with a Field-Programmable Gate Array (FPGA) for application acceleration, are today a common facility in datacenters and clouds. FPGAs can deliver tremendous improvements in performance and energy efficiency for a range or workloads, but development and deployment of FPGA-based applications remains cumbersome, leading to recent work which replicates subsets of the traditional OS execution environment (virtual memory, processes, etc.) on the FPGA.

In this paper we ask a different question: to what extent do traditional OS abstractions make sense in the context of an FPGA as part of a hybrid system, particularly when taken as a *complete package,* as they would be in an OS? To answer this, we built and evaluated Coyote, an open source, portable, configurable “shell” for FPGAs which provides a full suite of OS abstractions, working with the host OS. Coyote supports secure spatial and temporal multiplexing of the FPGA between tenants, virtual memory, communication, and memory management inside a uniform execution environment. The overhead of Coyote is small and the performance benefit is significant, but more importantly it allows us to reflect on whether importing OS abstractions wholesale to FPGAs is the best way forward.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are now standard in datacenters and cloud providers [1, 3, 12], providing more flexibility at lower power than ASICs or GPUs for many applications (e.g. [5, 19, 25, 29, 30, 41, 53]) despite (due to their heritage in embedded systems and prototyping) remaining difficult to program, deploy, and securely manage. As a result, along with much research into making FPGAs easier to program [7, 8, 36, 45, 51, 54, 58], 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, focusing on a particular aspect of functionality, e.g. Feniks [63] targets FPGA access to peripherals, Optimus [32] provides access to a host’s virtual memory via address translation, etc. These yield substantial incremental improvements over the 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 has a limited shelf-life, and so on.

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 have built Coyote, combining a coherent set of OS abstractions in a single unified runtime for FPGA-based applications. Like a microkernel, Coyote provides the core set of essential functions on which other services can be based: a uniform execution environment and portability layer, virtual memory, physical memory management, communication, spatial and temporal scheduling, networking, and an analog of software processes or tasks for user logic. It achieves this with minimal overhead (less than 15% of a commodity FPGA). Our contributions in this paper are therefore:

1. For a range of OS abstractions, a critical assessment of how each might map to an FPGA, in the context of its interaction with the others,
2. An implementation of the complete ensemble in Coyote, a configurable FPGA “OS” for hybrid compute servers.

3. A quantitative evaluation of Coyote using both microbenchmarks and 5 real applications.

4. A qualitative discussion of the implications of the work for future FPGA and OS designs.

We start with the basic hardware that any FPGA OS must handle. This determines the high-level structure of Coyote.

2 Foundations

Coyote targets hybrid systems, combining a conventional CPU with an FPGA either over a peripheral bus like PCIe, CXL [16], CCIX [13] or OpenCAPI [49], or instead a native coherency protocol as with Intel HARP [39] or ETH Enzian [2, 21]. Coyote runs today on PCs with Xilinx VCU118 [56], Alveo U250 [59] and Alveo U280 [60] cards. The port to Enzian is under way. We avoid any design decisions that might prevent the use of modern FPGA features like dynamic partial reconfiguration of multiple regions, or useful "hard" on-chip functions.

This naturally splits any design into a "hardware" component running on the FPGA and a software component running on the host CPU as part of the OS and support libraries.

Furthermore, 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, hardware components interact via standard interconnects like AXI [31].

2.1 The static region

The FPGA static region must contain the functionality required to reconfigure the dynamic region and communicate with the CPU's OS. However, its contents should not be fixed for all time. 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, different models of FPGAs show very different tradeoffs. In the medium term, it is important to make some static region components (for example, the TCP/IP stack) optional so they can be omitted if the space is better used for user logic.

In Coyote, the static region always contains logic to partially reconfigure the dynamic region, communicate with the host machine (an xDMA copy engine [57]), 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 (described below).

The static region can also contain optional logic shared between all applications running in vFPGAs, the most basic being memory controllers (for RAM directly connected to the FPGA) and networking (at present, TCP and RDMA).

2.2 The dynamic region

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 [3] and Intel's HARP [39]) dedicate this region to a single application, and reprogram it only rarely (e.g. when an associated virtual machine on the host is booted up).

Coyote, like other recent systems [14, 17, 62, 63], provides flexible spatial and temporal multiplexing. The dynamic region is partitioned into independent vFPGAs. Their number is wired into the static region, which allows multiple applications to run concurrently and be switched in and out.

A novel feature of Coyote is that each vFPGA is further divided into *user logic* and a *wrapper*. The former is a bitstream entirely synthesized by a Coyote user and validated by the system. This allows great flexibility in programming models: Coyote applications can be written in HLS, Verilog, VHDL, OpenCL, or some combination of these or other languages.

The wrapper is part of Coyote, and both sandboxes user logic and provides a standard interface 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, 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).

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 region), services (optional static components), system libraries (dynamic wrappers), and applications code (user logic). However, this would be an error. For example, the dynamic wrappers form part of a trusted computing base (TCB), whereas system libraries in a microkernel do not.

2.3 The software component

In a hybrid system, the host OS 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

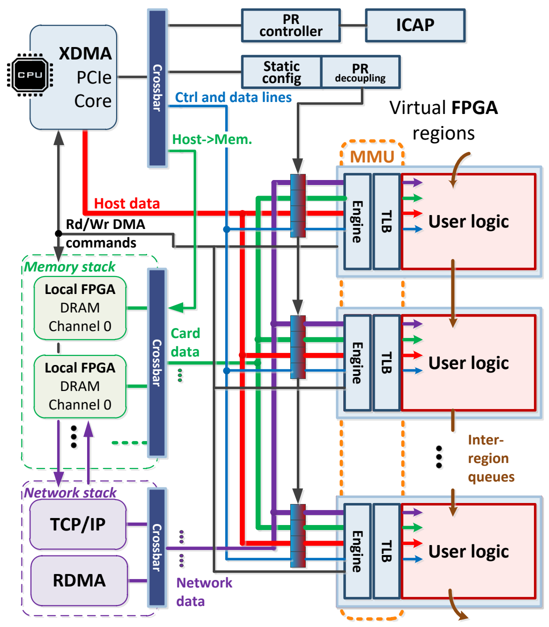


Figure 1: Coyote structure

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 maximises the FPGA area available to user logic, moving much functionality not on the fast path into the host CPU's OS. The software part of Coyote consists of a kernel *driver* (currently for Linux), a *runtime manager* process, and user application code.

At startup, the driver reads the configuration of the static region from the FPGA and sets up data structures for the set of vFPGAs to be supported. Thereafter, it is responsible for "control plane" communication with the FPGA (such as reconfiguring a vFPGA) and creating memory mappings for application code to interact directly with a vFPGA. The driver also handles dynamic memory allocation for the FPGA, and services TLB misses on the FPGA (see below).

Figure 1 shows the components of Coyote. The host CPU is connected to the PCIe core at top left.

3 OS abstractions on an FPGA

In this section, for each considered OS abstraction we first review its role in a conventional OS for a homogeneous multicore machine. We then discuss what is fundamentally different in an FPGA environment, and the impact this has on design decisions when creating an analog of the abstraction on the FPGA. Following this, we discuss our own implementation, and discuss our experience with building and using the approach. A quantitative evaluation of the whole of Coyote is given in Section 4.

3.1 Processes, tasks, and threads

The basic abstractions most OSes provide for multiplexing and virtualizing processor resources are based on processes, threads, and/or tasks. Definitions vary from OS to OS, but a thread is generally an open-ended execution of a sequence of instructions on a single virtual processor, a task is a unit of computational work to be dispatched to a CPU core, and a process is some combination of threads sharing an address space, 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 [9], and this is the foundation for programming models like CUDA and OpenCL. The task abstraction has also been successfully deployed on GPUs [44].

What's different on an FPGA? Resource multiplexing on FPGAs is fundamentally different, since there is no hardware corresponding to a "processor", "core", or "hardware thread" on which to base an abstraction aimed at multiplexing processing resources. Instead, the basic mechanisms available on the FPGA for multiplexing compute resources between principals are partial reconfiguration of areas of the FPGA logic at runtime, and spatial partitioning of application logic across different areas of the chip.

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.

An alternative proposed in research systems (e.g. [32, 62, 63] and others) is to partition the FPGA resources statically between applications. Spatial partitioning also raises further questions. For example, when multiple applications share the FPGA, should they be allowed to communicate, as processes do with IPC, and if so, how?

Coyote approach: Coyote combines both approaches, providing a cooperative multitasking abstraction of a set of virtual FPGAs, each of which is timeshared between applications. A Coyote platform is configured at boot time with a fixed number (e.g. 2-8) of vFPGAs, which are a spatial partition of the dynamic region of the chip. Each of these regions are, for the purposes of executing user logic, equivalent (much like cores in a symmetric multiprocessor), and are time-shared between applications. Ideally, a single application bitstream could be loaded into any available "slot" to be executed. Although some research in this direction exists [20], this is difficult with current levels of heterogeneity in FPGAs, which means that (at present) each application has to be (automatically) synthesized in advance for each vFPGA slot, akin to compiling "fat binaries" for multiple architectures. We discuss specific spatial and temporal scheduling questions below, along with the execution environment provided to user logic.

Discussion: A scheme with this generality requires care to implement. When timesharing vFPGAs, it is important that the context switch overhead does not outweigh the performance benefits of using a circuit in the first place. Dynamic partial reconfiguration of an FPGA is a relatively slow process and may remain so for the foreseeable future. In 4.3 we measure this cost.

Moreover, the logic required to implement multiple vFPGAs, and allow them to communicate and share services in the static region of the chip, must come with an acceptably-small cost in chip resources. We evaluate this in Section 4.2. So far our experience has been good: we can comfortably run multiple useful applications on a single FPGA today, and hardware trends are in our favour as the parts become larger.

3.2 Execution environment

The process abstraction also serves the purpose of providing a standard *execution environment* for a program. A program compiled to run in a process can, in principle, execute in any process on any machine implementing the same process 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.

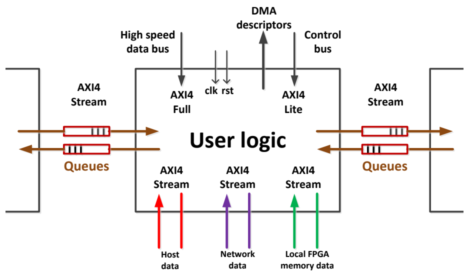


Figure 2: User logic interface

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. The same model has been implemented for FPGAs [51, 58].

However, hybrid FPGA-based systems are not pure computational devices - for example, they perform I/O through network and storage interfaces; indeed, this ability to interface externally is a major 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 [44], which present the GPU as a task-based runtime as opposed to a language-level OpenCL interpreter.

Coyote approach: Coyote defines a single *user logic interface* (ULI) for every application, which is the hardware analog of an ABI, and is illustrated in Figure 2. It uses the streaming AXI4 protocol for transferring bulk data between the host, memory stack, other services like the network stack, and the user logic. The same interface is used for inter-region communication, with a control plane over an AXI4-light bus.

This interface is provided by the dynamic 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 user logic at a fairly low level, allowing read and write descriptors to be generated directly from the user logic in the FPGA fabric, and host software access (including by high-bandwidth SIMD instructions) to be routed directly to the user logic.

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

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. For example, we ported publicly-available TCP and RoCE stacks [46, 48] to Coyote, and they became immediately usable to our existing applications without the extensive hardware-specific modifications usually required in FPGA development.

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 network stack to an ASIC (albeit off-chip). Intel's Embedded Multi-Die Interconnect Bridge (EMIB) is intended to extend FPGAs with new hardware, for example machine learning accelerators [37]. Recent Xilinx Versal cards also provide numerous off-chip hardware functions.

These trends make it even more difficult to achieve portability without a uniform execution environment like Coyote's to abstract these features behind a clean interface.

3.3 Scheduling

Scheduling on conventional machines is a complex topic with a history older than computers themselves. In this paper we focus on factors affecting scheduling mechanisms rather than specific 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. Some progress in this direction has been made in academia [27], but with significant performance penalties and implementation difficulties. Furthermore, the "state" of executing user logic potentially includes any stateful logic block (block RAM, flip-flops, DSPs) in the region of the FPGA used by the application, making the state capture all the more complex. For this reason, mechanisms for preempting arbitrary FPGA applications so that they can be reliably resumed later are not clear.

Instead, existing approaches to timeshare an FPGA avoid preemption [50] 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 [44]. 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.

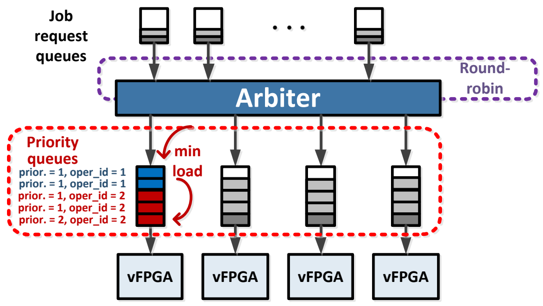


Figure 3: Coyote scheduling

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 job objects. Tasks are not scheduled by the FPGA itself, instead the runtime manager on the host CPU schedules them 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 3). 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 [32], 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: For the 5 applications we evaluate in Section 4, the reduction in the number of required reconfigurations substantially improves efficiency, to the extent that, with current hardware, it probably dominates other aspects of the scheduling algorithm. However, we feel there is still important work to be done in improving fairness, starvation-freedom, and predictability of the scheduler.

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 [26,32] 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 [33]. 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 stack (Section 3.6) and physical memory management (Section 3.5) 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.

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 address. 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 [4], copy-on-write, and transaction management [35] to dynamic binary translation [10].

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 [11,14,17,26,27,39,62,63]. 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 [55].

One approach to accessing host virtual memory from the FPGA is via the host platform's IOMMU. However, IOMMUs are not well-suited to a dynamic set of FPGA applications, even the subset that only use PCIe as an interconnect. Optimus [32] has a good explanation of the limitations of IOMMUs, and employs an ingenious "page table slicing" technique to work around them. Other recent work also implements some form of translation on the FPGA from user logic to host-physical addresses [6,15,42,52], .

These approaches, however, are limited to one special case: user logic accessing data on the host CPU memory in a software virtual address space. Modern FPGA platforms, however, have additional extensive memory closely coupled to the chip (for example, Enzian's FPGA has 512 GiB of DDR4), and also devices (such as network or storage controllers). To interact correctly with other OS abstractions (such as device virtualization, isolation, or even simply access to FPGA resources from the CPU), a virtual memory abstraction for multi-tenant FPGAs must thus be extended to these resources as well. It must enable safe and securely access to memory both on the FPGA itself, and on memory and devices directly attached to the FPGA, from user logic *and* software.

An approach satisfying these requirements is present in modern GPUs [38] where a unified memory abstraction of GPU and host memory is implemented. This abstraction provides primarily increased programmability which removes the need for explicit memory management and data movement from the software side.

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 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 [32] 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.

Coyote approach: Rather than relying on a single shared FPGA MMU, and/or relying on an IOMMU, Coyote includes TLBs 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 attached to the FPGA's memory controllers, 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 each dynamic region 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 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: Coyote actually provides two TLBs per vFPGA, one for 4KiB pages and one for 2MiB large pages. The TLBs are fairly straightforward caches (see Figure 4); associativity and number of sets are determined at build time for the application. A TLB miss causes an interrupt to the host CPU, whereupon the driver identifies the faulting vFPGA and either loads the TLB with a valid mapping or signals a page fault, which would be handled in software on the host.

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. Accesses to host and FPGA memory use different paths to proceed in parallel.

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. Dynamic wrapper registers 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 (see Figure 2) to be interpreted as the user logic sees fit.

Usage models: The most common way of using this facility is to provide GPU-style "Unified Memory", which 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. Coyote can thus handle multiple application contexts maintaining different shared virtual address spaces. It is the job of the software component of Coyote to ensure that address mappings are consistent between the vFPGA TLBs and the virtual address space of the corresponding software processes. Though there is no fundamental need for them to be so, it allows direct sharing of pointer-rich data structures between application software and user logic.

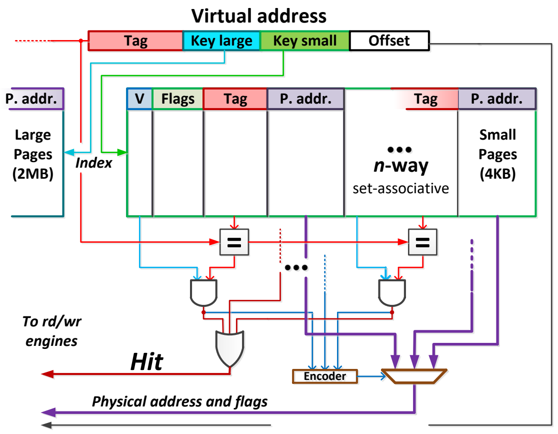


Figure 4: Coyote per-application TLBs

Alternatively, TLB entries can indicate that, when a corresponding virtual address is requested, the physical access is directly routed to either host or FPGA memory without any copying of pages. For efficient random access (such as pointer-chasing) this may be much more faster to program and execute than "unified memory".

Finally, it is also possible to route 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.

Discussion: As we show in Section 4, the TLBs impose very little space overhead in a vFPGA, 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 the software-loaded TLBs.

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.

What's different on an FPGA? Since FPGAs are much closer to the hardware, the situation is very different. Code running on FPGAs can access memory controllers directly. Data paths are not limited in size to cache lines, machine words, or MMU pages. SDAccel [58] exposes memory controllers explicitly to the programmer, providing flexibility but sacrificing simplicity and portability across FPGA devices.

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 (HBM) as well.

Moreover, as with servicing TLB misses, it may be useful to offload the dynamic allocation of FPGA memory to software, although hardware allocators have been developed [61].

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.

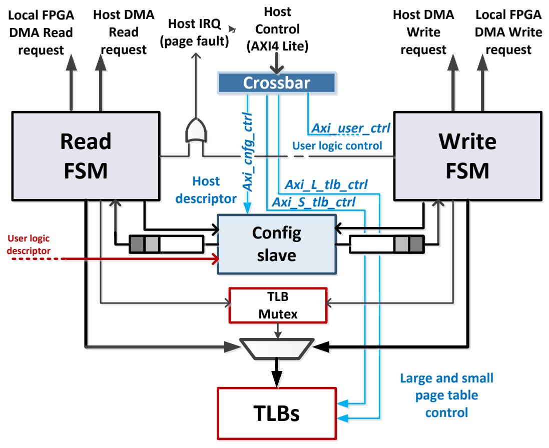


Figure 5: Read/Write engine

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 complexity of routing memory accesses originating from both user logic and host software, and destined for host memory or FPGA resources, is handled by a *read/write engine* in each vFPGA wrapper (Figure 5), which provides this flexibility in 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 the user logic issues them directly. This results in low overhead operations in the ULI entirely on virtual addresses. 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. Coupled DRAM is allocated in 2MiB superpages. 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 (as we show in Section 4.5) results in considerable performance gains over the naive approach. Accesses from different vFPGAs are still interleaved at each memory controller, as shown in Figure 6.

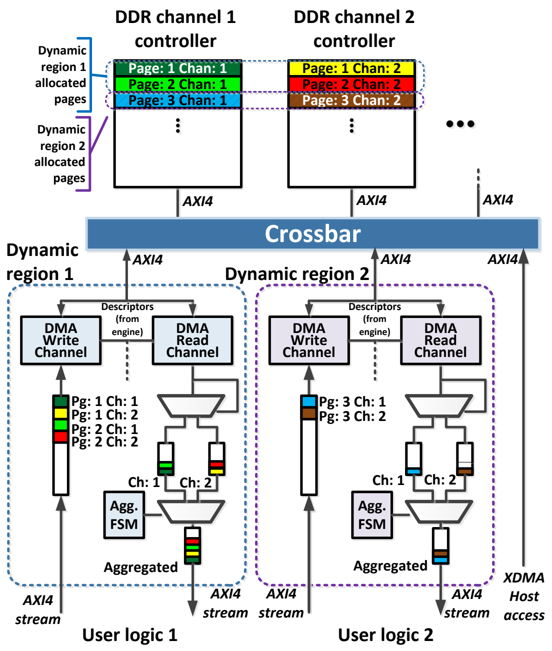


Figure 6: Multi-channel striping

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 (as we discussed in Section 3.2).

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.

3.6 IPC, I/O, and other services

A traditional OS provides a number of abstractions beyond those we have covered here. The most fundamental, at the heart even of microkernel architectures, is inter-process communication (IPC). 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 [42]. This allows users e.g. to chain dataflow operators running in different vFPGAs together while preserving the isolation between them.

While inter-vFPGA queues (and shared locations in FPGA virtual memory) can be used for inter-application communication, we find they are rarely used as such. 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.

Instead, the main use for such queues is communication with *services* provided by Coyote.

For example, Coyote provides an optional, but fully integrated, high-performance *multi-tenant* network stack based on our open-source TCP/IP and RDMA engine for FPGAs [46,47]. Like the memory stack described in Section 3.5, the network stack abstracts away the details of the physical network interfaces present and exposes a portable, standard interface, and can be shared between all vFPGAs present.

Further services can be similarly implemented and configured into the static region at startup, for example a storage stack (perhaps driving directly attached Flash memory). 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.

4 Evaluation

We focus on the question of whether the qualitative benefits of using Coyote's OS-style abstractions (scheduling, virtual memory, etc.) incur an acceptable quantitative cost in performance or efficiency. We look at overhead and space costs, as well as fairness in sharing resources, and the benefits of some of the optimizations in Section 3.

The hardware used for the results we report on here is a Xilinx VCU118 board [56] with an Ultrascale+ VU9P FPGA, attached to a host PC via PCIe x16. This interface provides a maximum theoretical bidirectional bandwidth of 16GiB/s. The board has 2 external DDR4 banks connected to the FPGA. Each DDR channel has a soft core DRAM controller instantiated in the FPGA fabric providing a total theoretical bandwidth of 18GiB/s. The host PC is a quad-core Intel i5-4590 at 3.3 GHz with 8GiB of RAM running at 1600MHz.

Unless stated otherwise, the system frequency used on the FPGA is 250 MHz. While each Coyote vFPGA has a separate PLL-generated clock, all the experiments reported here used the same frequency for the vFPGAs.

4.1 Macro-benchmark: decision trees

We first compare the performance of a complete, mature application running on Coyote with that obtained on Amazon F1 instances with the Xilinx SDAccel programming framework [58] and the Intel HARP environment [39]. It is hard to draw detailed conclusions from such a coarse-grained comparison, but we aim to show that (1) Coyote is a viable platform

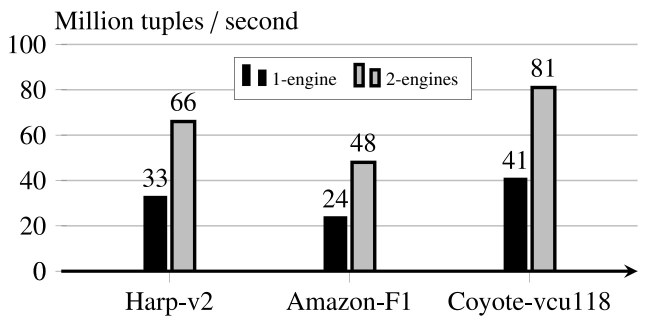


Figure 7: Performance of decision trees.

for real applications, and (2) the portability and programming features of Coyote come with negligible performance cost. The application is an open-source implementation [40, 41] of Gradient Boosting Decision Trees [34], focussing here exclusively on inference over decision tree ensembles.

Decision trees are a popular form of supervised machine learning for widely used tasks like classification and regression. They are constructed by recursively splitting the data into multiple groups. Using a cost function splits are evaluated and a greedy algorithm decides which split is the best candidate. Recursive splitting terminates at a pre-determined tree depth to prevent overfitting.

To do inference on the FPGA, the tree model is first loaded into FPGA on-chip memory. Data is then fetched from the host, inference performed, and the results copied back to host memory. All three phases are overlapped, allowing computation latency to be hidden behind memory operations.

We compare inference throughput (scored tuples per second) over Coyote with the same application on F1, and with a port running on Intel's hybrid CPU-FPGA HARP v2 platform. The latter is a rather different platform and the FPGA is clocked at 200MHz instead of 250MHz. Since F1 targets OpenCL applications, the SDAccel port employs a strict GPU-based compute model which incurs high data transfer overhead. In all cases, we measure throughput with both one and two instances of the application running on the FPGA with the data size of 4k tuples. On all platforms, the inference engine is compute-bound and requires only 4 GiB/s of memory bandwidth, allowing two instances to operate at full capacity.

The results are shown in Figure 7. Coyote provides comparable or better performance to that of the two commercial baselines. In the case of F1, this is despite Coyote providing portability and supporting multiple vFPGAs (SDAccel only allows a single dynamic region on the FPGA). True comparison with HARP is more tentative, given the lower clock frequency and very different hardware.

Nevertheless, we can conclude that, at the very least, there is no performance penalty in using Coyote in this case, and benefiting from the qualitative value it brings.

# vFPGAs	Stacks	LUTs	BRAM	Regs
1	✗	4%	4%	2%
2	✗	5%	5%	3%
4	✗	6%	7%	4%
1	✓	9%	10%	6%
2	✓	11%	12%	7%
4	✓	14%	14%	9%

Table 1: Resource overhead

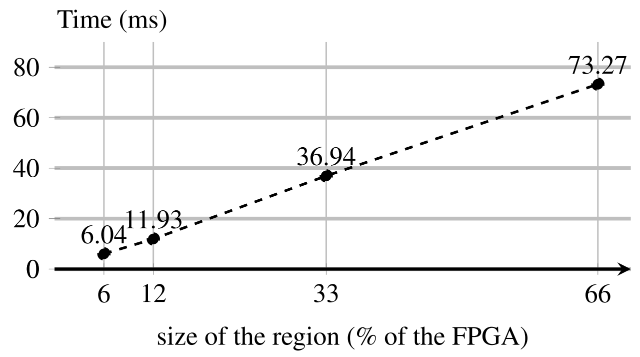


Figure 8: Time to load the operator and provide the view.

4.2 Space overhead

Raw performance is not the only consideration when comparing FPGA implementations, however. The space overhead (more precisely, the various resources on the chip used for the framework) can be just as important.

In this regard, Coyote (or any such set of abstractions) is strictly worse than a custom, native implementation of an application that takes over the whole FPGA, just as a bare-metal program is likely to use fewer resources than one running on top of Linux or Windows.

The space overhead of the framework for varying numbers of virtual FPGA regions and configurations with and without memory and network stacks are shown in Table 1. We give figures for the principal logic resources on the FPGA: lookup tables (LUTs), block RAM (BRAM), and registers (Regs).

On the VU9P, Coyote incurs a base overhead of 2-4%, increasing by < 1% for each additional vFPGA. Adding network and memory stacks roughly doubles this, incurring at most 14% for a full-featured Coyote install with 4 vFPGAs.

Larger future FPGAs and migration of often-used functionality into hard IP are likely to reduce this overhead still further. We therefore consider this to be a modest penalty in return for the benefits Coyote offers.

4.3 Micro-benchmark: context switching

We next measure the performance penalty in context switching a vFPGA from one application to another.

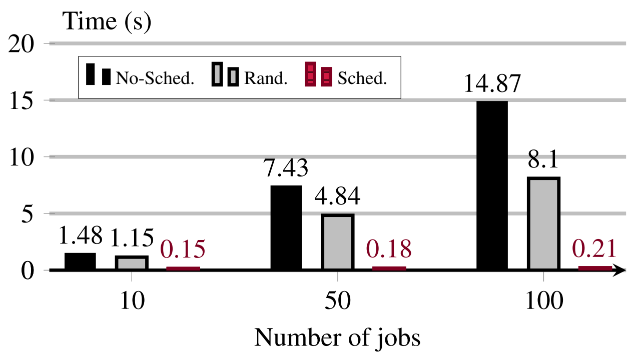


Figure 9: Scheduling performance.

Partial reconfiguration is still relatively slow on FPGAs, despite recent improvements. Figure 8 shows partial reconfiguration latency of the Xilinx VU9P as a function of the chip area to be reconfigured; Coyote with 4 vFPGAs would correspond to roughly 20%, or 20ms, per vFPGA.

We note that achieving even this performance is non-trivial. Coyote's implementation uses a fast data stream to the ICAP (the FPGA unit handling partial reconfiguration) which can saturate its bandwidth of about 800MiB/s. In contrast, SDAccel achieves a mere 19MiB/s over a slow AXI4-Lite link.

As mentioned in Section 3.3, this overhead can be reduced substantially by sharing the same vFPGA logic between successive tasks targeting the same partial bitstream (compute operator). We illustrate this with a simple example by scheduling queues of tasks from the four applications used in Section 4.4, all with small transfers of 4KiB each and all with the same priority. We configure Coyote here with 3 vFPGAs.

We dispatch jobs in three ways: `no-sched` is round-robin in both queues and vFPGAs. This causes a reconfiguration for each job and is the worst-case scenario. `rand` picks the next job from a random queue each time, and so has a 1-in-3 chance of needed a reconfiguration, and `sched` uses Coyote's heuristic of grouping jobs which share the same user logic.

Figure 9 shows total turnaround time for 10, 50, and 100 jobs of each type. Unsurprisingly, minimizing the number of partial reconfigurations has a dominating effect on total system throughput. Clearly there is room here for much more sophisticated job scheduling, beyond the scope of this paper.

4.4 Resource sharing

We now evaluate how the OS-like features of Coyote can provide fair sharing of resources across multiple vFPGAs simultaneously. In cloud deployments, stable and predictable distribution of resources is a key requirement.

We run four different applications on Coyote: AES encryption, sha256 hash computation, HyperLogLog multiset cardinality estimation [18, 28] and k-means calculation [22]. In each experiment we run an application with one or more simultaneous vFPGA instances at a time. All tests except the k-means are using the host memory and direct streaming. Due to the iterative nature, the k-means utilizes accesses to the local FPGA memory.

We measure per-application round-trip throughput vs. transfer size, including the transfer of the plain data to the FPGA user logic, pipeline computation, and simultaneous transfer of the computed results back to the memory. We use hardware counters in the FPGA fabric and so incur no overhead.

When multiple applications are running concurrently, we also calculate mean absolute deviation (MAD) of the instances from the average performance results, to give a quantitative measure of (un)fairness in resource allocation.

Results are shown in Figure 10. sha256 (Figure 10.a) is compute bound, and performance scales perfectly as long as all the vFPGAs fit in the FPGA.

More interesting is AES (Figure 10.b) which is memory-bound. Here multiple AES vFPGAs are competing for PCIe bandwidth, which is saturated in all cases due to the AES implementation being heavily pipelined. We observe that, firstly, throughput of an AES instance is inversely proportional to the number of peers in the system, showing that the scarce resource of PCIe bandwidth is being shared between them, and also the MAD is very low compared with total bandwidth, suggesting that sharing is fair.

The HyperLogLog implementation uses 16 parallel pipelines that are able to compute on a single cache-line at a time. The module is thus able to sustain processing at line rate for larger transfers (as are mostly present during cardinality estimation). For smaller transfers processing latency is the domineering factor. The results are shown in the Figure 10.c.

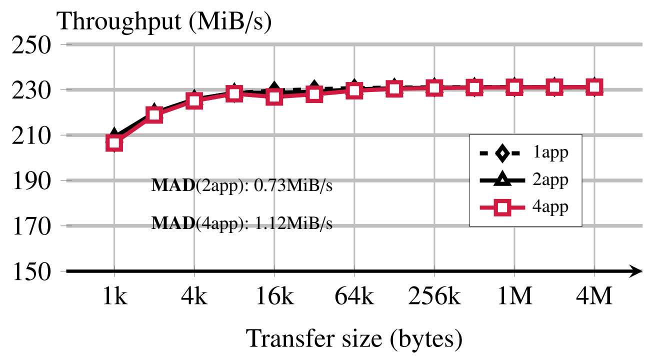
The final results (Figure 10.d) show the throughput of the k-means clustering operator during the single computation iteration. This is an iterative algorithm, where data is first offloaded from the host to the local FPGA memory. The data is then streamed to the user logic in each iteration and dispatched to 16 parallel pipelines on the FPGA to compute centroids, the results of which are then transferred back.

In summary, these results validate our goals of sharing scarce bandwidth on the FPGA between multiple tenants.

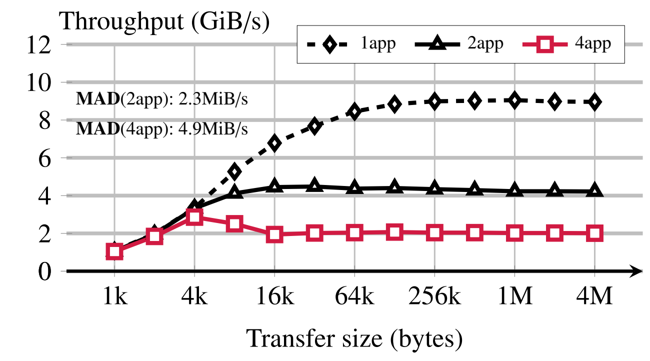
4.5 Striping

We evaluate the impact of Coyote hiding individual DRAM channels behind a single abstraction that stripes each 2MiB page across all channels on the FPGA for portability.

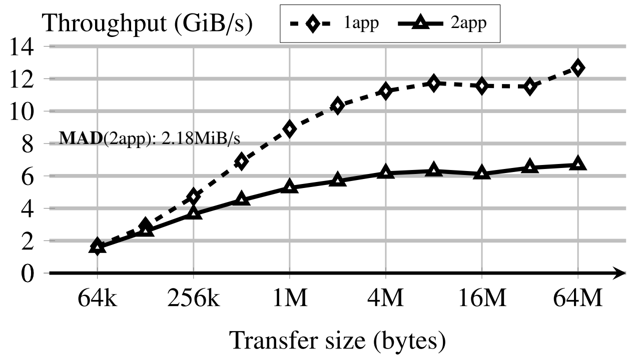
The benchmark is a simple DRAM to DRAM copy, implemented entirely on the FPGA and measuring throughput for transfers ranging from 4KiB to 1MiB. We measure bandwidth for three scenarios. First, `1-channel` copies memory using on a single channel, and is the baseline for performance. Second, `2-channel` reads from one channel and writes to another. This is the best case and requires knowledge of all the channels in the FPGA. It could be achieved in, for example,



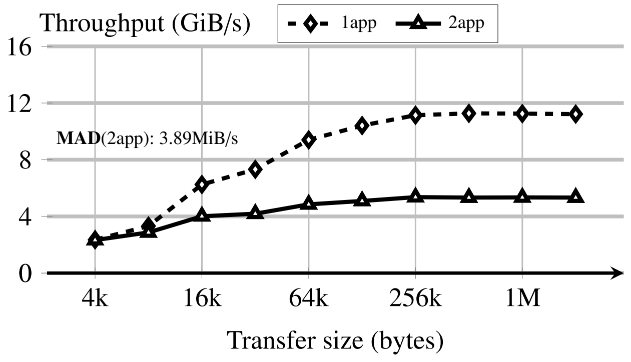
(a) sha256 round throughput for different number of concurrent applications in dynamic regions.



(b) AES round throughput for different number of concurrent applications in dynamic regions.



(c) HyperLogLog throughput.



(d) k-means single iteration throughput.

Figure 10: Performance benchmarks for example applications running in Coyote showing fair sharing of the bandwidth.

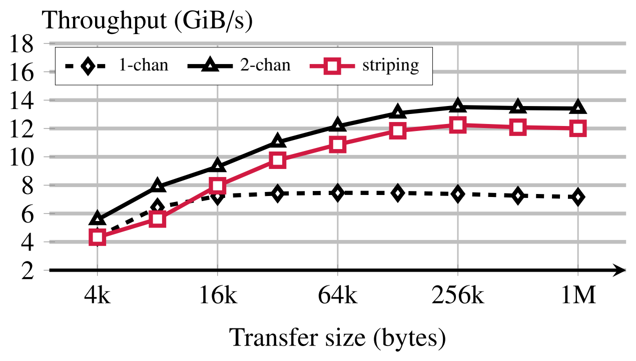


Figure 11: Striping performance.

SDAccel by explicit placement of data on the channels and careful FPGA-specific code. Finally, `striping` shows Coyote's performance when oblivious to channels and placement, and each data page is striped across channels.

Figure 11 shows the results. For small transfers, setup costs dominate, but a single channel becomes saturated at 16KiB and two channels at about 128KiB. Coyote incurs an overhead of about 10%, which is competitive with many cases of hand-optimized vs. compiler generated software code, and leads us to conclude that abstracting the DRAM controllers is a worthwhile trade-off for performance isolation and portability.

4.6 Demand paging

GPU-style "unified memory" implements a form of distributed shared virtual memory between the host and FPGA, largely abstracting memory management and explicit data movement from the users. When a vFPGAs tries to access virtual locations on the local FPGA memory which are not present in the physical memory, a page fault is generated and the driver initiates a copy from host to FPGA memory, then adjusts page tables on both sides, before signalling the vFPGA that it can proceed.

Without this model, explicit copying of data would be required, as illustrated by this pseudocode:

```
void* host_d = malloc(size);
void* fpga_d = getFpgaMem(size);
offloadMemCpy(host_d, fpga_d, size);
executeOperator(fpga_d, size);
free(host_d);
freeFpgaMem(fpga_d);
```

The demand paging provided by "unified memory" allows a simpler (for the programmer) model, resulting in code like this:

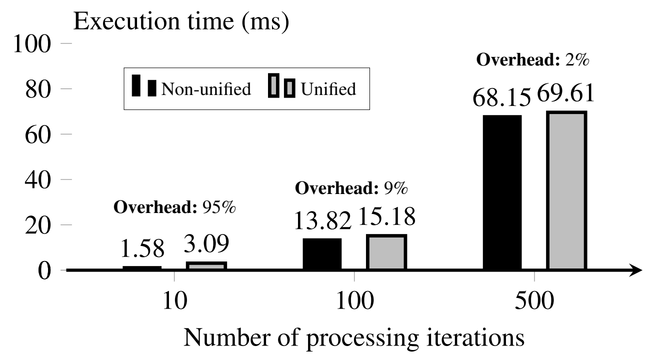


Figure 12: Unified memory overhead.

```
void* host_d = malloc(size);
executeOperator(host_d, size);
free(host_d);
```

In a fully cache-coherent system like Enzian [21] this code would not require copies at all, but in PCIe-based systems they are needed in both directions: after the computation has completed, the pages holding the computation results must also be copied back to the host physical memory.

The cost, therefore, of the unified memory abstraction stems from page faults on both sides, remapping pages by modifying page tables, and copying the data between host and FPGA. In practice, this cost is amortized over the number of iterations (for example) that the computation performs for each transfer.

Figure 12 shows this overhead in the context of the whole computation. The workload represents kmeans iterating over 1MiB of data which has to be moved from the host to local FPGA memory. We vary the number of iterations and measure the impact of the page fault overhead and initial copy.

For a single iteration the overhead is high (95%), reflecting the fact that an iteration executes extremely quickly on the FPGA and is comparable in time to the copy, and suggesting that this model of memory usage is not ideal for streaming applications. However, as the iteration count reaches 500, the overhead has reduced to 2% and is likely to be an acceptable price to pay for programming convenience.

5 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 3, and two recent surveys [24, 50] give an excellent overview. In this section, therefore, we focus on a few important recent systems.

The initial version of Microsoft's Catapult [12, 43] 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 [39] turns the FPGA into one more processor. Intel implements its own QuickPath interconnect [23] 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 [58], used by Amazon [3] and Alibaba [1] 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 [26] 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 [32] 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 [62] 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 (like AmorphOS) 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-user TCP/IP stack, unified memory translation/protection across CPU and FPGA, inter-application communication, 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. We return to this point in our conclusion.

6 Conclusion

Coyote approaches the FPGA shell as an *operating system design problem*. While putting individual OS features on an FPGA has value, taking a holistic view allows us to identify how things fit together. The design of virtual memory on an FPGA changes radically when one takes into account e.g. FPGA-local devices, or the need to abstract local DRAM controllers. Conversely, abstracting such controllers only works when one has the right MMU design in place.

For example, allowing both software and hardware applications to initiate virtual memory accesses to both host and FPGA memory resources enables a uniform execution environment and portability across different memory systems, but may rule out the application-implemented checkpoint-and-restore approaches ViTAL and Optimus use for cooperative "preemption", since there is now per-application state (TLBs, etc.) not accessible to user logic.

As FPGAs become larger, the demand for the traditional OS functions of secure multiplexing, sharing, and abstraction will grow. At the same time, so will the opportunity to provide more OS-like functions on the FPGA. It is important that these functions work together. Our evaluation shows that the price of this complete OS functionality is more than acceptable in throughput, space efficiency, scheduling overhead, and memory bandwidth.

A further hardware trend, moreover, is the migration of commonly-used functions out of synthesized logic and into hard IP cores on the FPGA. In this rapidly-changing landscape, the right set of abstractions can prevent hard-to-develop OS-style logic from becoming rapidly obsolete.

Experience with software also suggests that OS abstractions are “sticky”: once decided, they alter very slowly over time even when the underlying hardware changes radically, to the detriment of performance and security. This suggests that it is vitally important to get things “right” as early as possible.

Coyote is a small step in this direction, and shows that a coherent and reasonably complete set of OS abstractions, suitably modified, can map well onto an FPGA, deliver both immediate and longer-term benefits, and impose only a modest overhead on today’s hardware.

Coyote can be downloaded at https://github.com/fpgasystems/Coyote.

Acknowledgements

This work has been made possible through a generous equipment donation from Xilinx and through access to the Xilinx Adaptive Compute Cluster (XACC) Program. The research of Dario Korolija is funded in part by a grant from HPE. We would like to thank Mohsen Ewaida, Zhenhao He and Amit Kulkarni for some of the use cases and designs, David Sidler for feedback and help with the network stack, the anonymous reviewers for their helpful comments and questions, and our shepherd Baris Kasikci for feedback and guidance.

References

[1] Alibaba Cloud Services. Compute optimized instance

families with FPGAs. https://www.alibabacloud.com/help/doc-detail/108504.htm, May 2020.

[2] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Owaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. Tackling Hardware/Software co-design from a database perspective. In *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands, January 2020.

[3] Amazon Web Services. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/, May 2020.

[4] Andrew W. Appel and Kai Li. 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, 1991. Association for Computing Machinery.

[5] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131, Aug 2009.

[6] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access*, 5:1900–1910, 2017.

[7] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *ACM Queue*, 11:40:40–40:52, 2013.

[8] Donald G. Bailey. 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, 2015.

[9] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A `fork()` in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, Bertinoro, Italy, May 2019.

[10] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the X86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4), November 2012.

[11] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 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, 2014.

[12] 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. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, October 2016.

[13] CCIX Consortium and others. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com, January 2019.

[14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 3. ACM, 2014.

[15] Eric S. Chung, James C. Hoe, and Ken Mai. 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, 2011.

[16] CXL Consortium. Compute Express Link. https://www.computeexpresslink.org/, May 2020.

[17] Suhaib Fahmi, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE, 2015.

[18] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In Philippe Jacquet, editor, *AofA: Analysis of Algorithms*, volume DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of *DMTCS Proceedings*, pages 137–156, Juan les Pins, France, June 2007. Discrete Mathematics and Theoretical Computer Science.

[19] D. Fortún, C. G. de la Cueva, J. Grajal, M. López-Vallejo, and C. L. Barrio. Performance-oriented Implementation of Hilbert Filters on FPGAs. In *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, Nov 2018.

[20] B. Gottschall, T. Preußer, and A. Kumar. 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, April 2018.

[21] ETH Zurich Systems Group. Enzian, a research computer. http://enzian.systems, May 2020.

[22] Zhenhao He. Bit-Serial kmeans. https://github.com/fpgasystems/bit_serial_kmeans, October 2020.

[23] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, January 2009.

[24] C. Kachris and D. Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016.

[25] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. 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, April 2017.

[26] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, 2018.

[27] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. Migration of long-running Tasks between Reconfigurable Resources using Virtualization. *ACM SIGARCH Computer Architecture News*, 44(4):56–61, 2017.

[28] Amit Kulkarni, Monica Chiosa, Thomas Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. FPGA-based HyperLogLog Accelerator. https://github.com/fpgasystems/fpga-hyperloglog, October 2020.

[29] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb 2007.

[30] X. Li, L. Ding, L. Wang, and F. Cao. FPGA accelerates deep residual learning for image recognition. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 837–840, Dec 2017.

[31] ARM Ltd. AMBA 4 AXI4-Stream Protocol. https://developer.arm.com/docs/ihi0051/latest, 2010.

[32] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, September 1991.

[34] Alexey Natekin and Alois Knoll. Gradient Boosting Machines, A Tutorial. *Frontiers in neurorobotics*, page 21, 2013.

[35] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 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, 2015. Association for Computing Machinery.

[36] R. Nikhil. 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, June 2004.

[37] 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. 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, 2018.

[38] NVIDIA Corporation. *Unified Memory in CUDA 6*, version: 2.0, document revision: 29 edition, Nov 2013. https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[39] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. A reconfigurable computing system based on a cache-coherent fabric. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 80–85. IEEE, 2011.

[40] Muhsen Owaida and Gustavo Alonso. Distributed Inference over Decision Tree Ensembles. https://github.com/fpgasystems/Distributed-DecisionTrees, October 2020.

[41] Muhsen Owaida, Amit Kulkarni, and Gustavo Alonso. Distributed Inference over Decision Tree Ensembles on Clusters of FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 12(4), September 2019.

[42] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 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, 2017.

[43] 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. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[44] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 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, 2011. Association for Computing Machinery.

[45] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. https://arxiv.org/abs/1505.01120v1, 2015.

[46] D. Sidler, Z. István, and G. Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[47] David Sidler, Monica Chiosa, Zhenhao He, Mario Ruiz, Kimon Karras, and Lisa Liu. Scalable Network Stack supporting TCP/IP, RoCEv2, UDP/IP at 10-100Gbit/s. https://github.com/fpgasystems/fpga-network-stack.git, October 2020.

[48] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *EuroSys'20*, pages 29:1–29:16, 2020.

[49] J. Stuecheli, B. Blaner, C.R. Johns, and M.S. Siegel. CAPI: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1), 2015.

[50] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A survey on FPGA virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE, 2018.

[51] Malte Vesper, Dirk Kocha, and Khoa Phama. PCIeHLS: an OpenCL HLS framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 10–15. IEEE, 2017.

[52] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 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, 2018.

[53] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. *CoRR*, abs/1901.04988, 2019.

[54] Skyler Windh, Xiaoyin Ma, Robert Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid Najjar. High-Level Language Tools for Reconfigurable Computing. In *Proceedings of the IEEE (Volume: 103 , Issue: 3 , March 2015)*, pages 390 – 408. IEEE, 2015.

[55] F. Winterstein and G. Constantinides. 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, 2017.

[56] Xilinx. VCU118 Evaluation Board User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/vcu118/ug1224-vcu118-eval-bd.pdf, October 2018.

[57] Xilinx. 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, November 2019.

[58] Xilinx. *SDAccel Environment User Guide*, v2019.1 edition, May 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf.

[59] Xilinx. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*, v.1.3.1 edition, May 2020. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#documentation.

[60] Xilinx. *Alveo U280 Data Center Accelerator Card Data Sheet*, v.1.3 edition, May 2020. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#documentation.

[61] Zeping Xue and D. B. Thomas. 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, September 2015.

[62] Yue Zha and Jing Li. 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, 2020. Association for Computing Machinery.

[63] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 22. ACM, 2017.

A Artifact Appendix

A.1 Abstract

Coyote brings operating system abstractions to reconfigurable heterogeneous architectures. It provides a range of abstractions which ease the interaction between the host, the FPGA, the memory and the network. The following sections will describe the process of obtaining the framework resources and building them. The application deployment procedure is shown as well.

A.2 Artifact check-list

- **Compilation:** HLS, CMake, C++, Boost.
- **Run-time environment:** Vivado, Linux.
- **Hardware:** Xilinx.
- **Metrics:** Throughput, latency, resources, reconfiguration.
- **Experiments:** HyperLogLog, kmeans, AES, sha256, decision trees, microbenchmarks.
- **Required disk space:** 4MiB.
- **Expected experiment run time:** 2 hours.
- **Public link:** `https://github.com/fpgasystems/Coyote`.

A.3 Description

A.3.1 How to access

The open-source version of the Coyote framework can be found on Github at the following address:
`https://github.com/fpgasystems/Coyote`.

A.3.2 Hardware dependencies

The framework targets a variety of Xilinx data center and development boards. At this point full support for the following boards is provided: Alveo U250, Alveo U280, VCU118. The boards have to be attached to the host system over the PCIe. If available, the framework takes advantage of the AVX2 (*Advanced Vector Extenstions*) instruction set. Legacy support is also provided.

The hardware build process relies on the Vivado toolchain and its high-level synthesis extension. Versions 2019.2 and 2020.1 have been officially tested. The toolchain is used for the compilation of the full and partial bitstreams. It also handles the deployment of the full bitstreams. The automation of the hardware build process is done with CMake. Minimum required version is 3.0.

A.3.3 Software dependencies

The software build process is split between the low level Linux kernel driver and the high level user application layers.

The driver code was tested on the machine with the Linux kernel version 5.4. This code is built with Makefile.

The user application layer is fully written in C++11. The Boost libraries (`https://www.boost.org/`) are used for the parsing of the command line arguments. As with hardware, CMake with a minimum version of 3.0 is required for the software build automation.

A.4 Installation

Pull the newest version of the repository:

```
$ git clone https://github.com/fpgasystems/Coyote
$ cd Coyote
```

The network stack submodule and the correct branch can then be initialized:

```
$ git submodule update --init --recursive
```

At this point all the necessary resources are available locally. Further build is split into separate hardware and software processes.

A.4.1 Hardware build

This section explains the process of creating the custom hardware design, the integration of the arbitrary user logic and finally the formation of the valid FPGA bitstreams.

First create the build directory inside the hw directory:

```
$ cd hw
$ mkdir build
$ cd build
```

Enter a valid chosen system configuration:

```
$ cmake .. -DFDEV_NAME=u250 <params...>
```

Large level of configuration flexibility for the framework is available. This allows the framework to adapt to a variety of processing scenarios. The following parameters can be chosen (bolded parameters are passed by default):

- **FDEV_NAME:** This is the name of the target device. Supported parameters are <**`u280`**, `u250`, `vcu118`>.
- **N_REGIONS:** This is the number of concurrent vFPGAs (independent regions). The maximum of 16 regions per FPGA is supported at the moment <**`1`**`:16`>.
- **EN_STRM:** Enables the direct host-FPGA streaming over PCIe lanes <`0`, **`1`**>.
- **EN_DRAM:** Enables the local FPGA memory stack <**`0`**, `1`>. It can work in conjunction with the streaming.
- **N_DRAM_CHAN:** The number of the chosen DRAM channels. The maximum available number depends on the target board. <**`1`**`:4`>.

- **EN_PR:** Enables the partial reconfiguration flow <`0, 1`>. This partitions the FPGA fabric into multiple dynamic regions. The number depends on the amount of vFPGAs present. A separate partial bitstream will be generated for each dynamic region. Manual floorplanning of dynamic regions is advised.
- **EN_TCP:** Enables the 100G TCP/IP stack <`0, 1`>. This integrates a TCP/IP network stack and exposes its communication interface to every vFPGA.
- **EN_RDMA:** Enables the 100G RDMA stack <`0, 1`>. This integrates a full RDMA network stack with reliable communication protocol (RC) built on top of RoCE v2. Interface is exposed to every vFPGA.

The build directory with the chosen configuration is initiated once the previous command completes. Now referenced high-level synthesis cores can be built:

```
$ make installip
```

The hardware project can then be created:

```
$ make shell
```

Once this command completes, the project with one static and initial vFPGA regions is created (config 0). If partial reconfiguration flow is enabled, additional sets of partial bitstreams (new logic for each vFPGA) can be created:

```
$ make dynamic
```

This command can be executed multiple times to create multiple sets of partial bitstreams (config 1, 2, 3, ...).

At this point the user logic can be inserted into vFPGAs. Wrappers can be found under build project directory in the hdl/config_X. Once the user design is ready to be compiled, run the following command:

```
$ make compile
```

When the compilation finishes, the initial bitstream with the static region can be loaded to the FPGA via JTAG. This can be done in Vivado's programming utility. At any point during the compilation, the status can be checked by opening the project in Vivado (`start_gui` command).

All compiled bitstreams, including partial ones, can be found in the build directory under bitstreams.

A.4.2 Driver

The driver can be compiled on the host machine:

```
$ cd driver
$ make
```

Once the bitstream is loaded on to the target FPGA, the rescan of the PCIe can be executed with the utility script:

```
$ ./util/hot_reset.sh
```

If during this card detection fails, warm reboot of the host machine has to be completed. The driver can then be inserted into the kernel:

```
$ insmod fpga_drv.ko
```

The software applications can now be executed.

A.4.3 Software build

This section explains the process of building the user applications that utilize the provided high-level API. Additionally, the scheduling example provides the runtime manager which abstracts the application deployment.

First create the build directory inside the directory of the chosen software project:

```
$ cd sw/<project>
$ mkdir build
$ cd build
```

Initiate the build configuration and compile the executable:

```
$ cmake ..
$ make main
```

System permissions need to be assigned to the executable.

A.4.4 Simulation

The user logic hardware can be simulated in Vivado:

```
$ cd hw/sim/scripts/sim
$ vivado -mode tcl -source tb.tcl
```

At this point any user logic can be inserted and arbitrary stimulus applied. The signal behaviour can then be observed.

A.5 Evaluation and expected result

The user logic for hardware applications (HyperLogLog, kmeans, AES, decision trees, sha256) and all microbenchmarks can be found under hw/hdl/operators. Examples of specific network operators are supplied as well. Default configurations of every operator coincide with ones used to obtain the results in the paper.

The code in sw/base is used for the tests where no explicit operator control is needed (AES, HyperLogLog, sha256). The code in sw/scheduling is used for the measurements of the reconfiguration time. Separate code for the operators requiring more control is provided (kmeans and decision trees).

A.6 Experiment customization

A wide variety of test cases and customization is available through different system configurations. The users can create different versions of the system through combinations of vFPGAs, network and memory stacks.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts