A Comprehensive Study of Coyote Memory System

Yuan Luo
Technical University of Munich

Abstract

FPGA has been a trend in modern data center and cloud computing. It can deliver tremendous performance improvements, including I/O throughput, power efficiency, and network speed. Programming FPGA has always been a challenge. Also, dealing with different I/O devices can be difficult as well. Coyote targets a hybrid system with CPU and FPGA. It provides OS abstractions to FPGA and an I/O scheduler for I/O transactions.

In this article, we will study Coyote memory system in more detail to build an efficient I/O scheduler. First, we talk about PCIe connectivity and the static and dynamic region of FPGA. They are the basis for memory abstraction and process abstraction. Then, we discuss how data transfers are carried out with the help of a unified memory and read/write engine in Coyote. Debugging FPGA in Coyote can be done using Integrated Logic Analyzer. Finally, two example programs are executed to show the host-initiated and FPGA-initiated data transfer.

1 Introduction

Field-Programmable Gate Array (FPGA) has now been widely deployed in datacenters and cloud services [4, 5]. It can provide lower latency than GPU [14] and higher network bandwidth than Network Interface Controller [10]. However, the lack of programmability and portability makes it difficult to program, deploy, and manage FPGAs.

Coyote [9] brings traditional OS abstractions to FPGA. It utilizes spatial partitioning and partial reconfiguration to enable the FPGA process. Coyote has also prepared the execution environment for FPGA processes accordingly. Other OS abstractions, such as scheduling, virtual address, and memory management, are also introduced by Coyote. Coyote also supports memory I/O and network I/O scheduling.

With Coyote providing I/O scheduling on FPGA, we can now try to build an I/O interface to abstract all I/O interfaces on FPGA (e.g., PCIe bridge, network I/O, and USB I/O) and an I/O scheduler to efficiently schedule all I/O transactions. The I/O scheduler shares some similarities with the state-of-the-art network packet scheduler on FPGA [15, 13], but should also consider the difference between network packet and I/O transaction. To achieve this, we need to know Coyote's technical concepts in more detail and how exactly an I/O request is processed in Coyote.

In this paper, we first discuss the software driver, which works with DMA Subsystem for PCIe (XDMA) to enable communication and collaboration between CPU and FPGA in Coyote. Then, we look at virtual address and memory management in Coyote. Since debugging FPGA can be quite different, some instructions on how to do it will also be provided. Finally, some test programs are provided to show and evaluate what we are discussing.

2 Coyote Architecture

2.1 System overview

Coyote targets the heterogeneous computing system consisting of CPU and FPGA as shown in Figure 1. Data and program execution are distributed in both CPU and FPGA, making the most of the benefits of different computing resources. To distribute and collect data, communication between CPU and FPGA needs to be established. To distribute the execution, the FPGA needs to be assigned with different logic circuits on the fly. To achieve this, Coyote is split 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. Figure 1 provides a simple overview of the system. CPU and FPGA are connected via PCIe. The driver abstracts FPGA resources to provide a simpler interface to access FPGA. Programs running on the host can interact with the FPGA through the driver, such as transferring data, assigning computation tasks to the

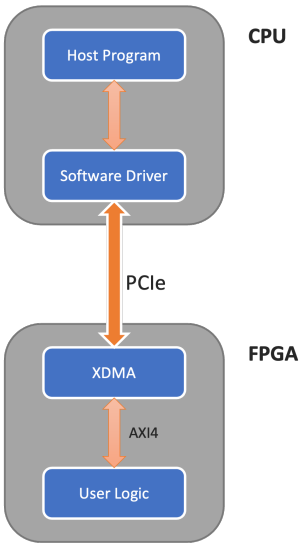

Figure 1: System architecture overview

FPGA, and allocating FPGA memory. FPGA can also remotely access some memory on the host with XDMA for data transfer.

2.2 Communication between CPU and FPGA

Communication between CPU and FPGA is based on the driver and XDMA. Figure 1 shows how CPU and FPGA are connected. On the CPU side, a software driver is built to manage FPGA. While on the FPGA side, XDMA IP is introduced to configure PCIe and establish Direct Memory Access (DMA).

For the Coyote driver, we ignore the details about how the driver is built on Linux, as [7] has already provided a comprehensive explanation of this. Instead, we focus on PCIe address spaces and Base Address Register (BAR), which are critical for the Coyote driver. The PCI address domain consists of three distinct address spaces: configuration address space, memory address space, and I/O address space [12]. The configuration address space contains BARs. BARs hold the base addresses of memory space or I/O space, and memory space and I/O space can be further subdivided into multiple segments. The address of each segment can be manually calculated with the base address stored in BAR. Memory address space and I/O address spaces are two address ranges that support different memory access instructions. They are used exclusively for data transfer. These addresses can be mapped to virtual addresses on the host side via function *pci_iomap()* so that the program on the host computer can easily communicate with the PCIe device by writing or reading data to these addresses.

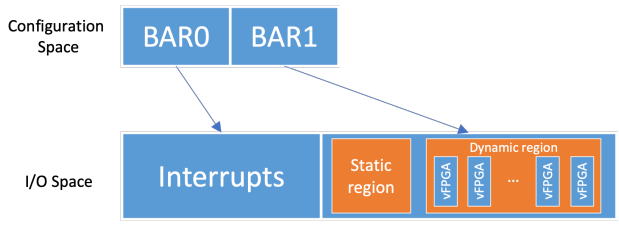

Figure 2: Configuration and I/O Space

XDMA is an IP for PCIe-based communication protocol provided by Xilinix [2]. One end of XDMA is a PCIe interface, which is connected to the PCIe slot on the host motherboard via the pins of FPGA. The other end of XDMA connects to the user logic on FPGA via AXI4 protocol [1]. The key component for Coyote in XDMA is the BARs. BARs in Coyote are 64 bit wide and only the first 2 out of 3 BARs are actually used. The first BAR contains addresses for memory regions to implement interrupts. The second BAR contains addresses for memory regions to configure static and dynamic regions. In addition, XDMA provides DMA for FPGA to access data residing in host memory.

2.3 Static region and dynamic region

FPGA is a Reconfiguration Hardware System (RHS), which can convert a portion of computationally intensive code into a hardware circuit depending on the task being performed [6]. An RHS is split into two regions: static and dynamic [8]. On FPGA, the static region consists of designs that do not need to be altered, such as XDMA mentioned above and the circuit to control the dynamic region. The dynamic region partially reconfigures itself into a circuit that performs the assigned computation when receiving control signals from the static region.

Static region in Coyote is set at compile time. It consists of fixed logic and optional logic. The logic to partially reconfigure the dynamic region, communicate with the host machine, and divide the dynamic region into a set of virtual FPGAs (vFPGAs) always resides in it. Other logics such as FPGA memory controller and networking can optionally be enabled. vFPGA is based on the idea of dividing dynamic regions into smaller regions. Each smaller region is called vFPGA. As Figure 2 shows, two BARs in the configuration space hold addresses in the I/O space. I/O space is divided into two ranges, one for interruptions and the other for static and dynamic regions. The dynamic region is further divided into several subregions. Each subregion is assigned a vFPGA.

Dynamic regions can be compiled at runtime. It works with partial reconfiguration to enable process abstraction on FPGA. Partial reconfiguration is a mechanism for exploiting FPGA resources in a spatial multiplexing

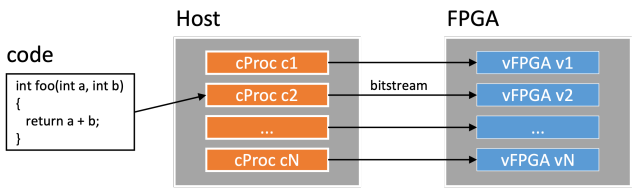

Figure 3: cProc and vFPGA

manner. PCIe memory address space for the dynamic region can be divided into several regions depending on how many vFPGAs are enabled, as shown in Figure 2. Each vFPGA can have its own memory address space on the host side. In this way, vFPGAs are abstracted to Coyote Process (cProc) on the host side so that they can be more properly managed by the driver and easier to use by programmers. cProc is a Coyote defined process. It provides methods such as to allocated memory on FPGA, to access Control and State Register (CSR), and to reconfigure vFPGA. With independent address space for each vFPGA on the host side and cProc abstraction, the programmer can easily program the FPGA without interacting with it directly.

Figure 3 shows the relationship between cProc and vFPGA. On the host side, the bitstream of the function to be assigned to the FPGA is given to a cProc. cProc will then load the bistream onto the FPGA and reconfigure it. When the vFPGA receives the bitstream, it reconstructs its circuit area based on the bitstream to implement the required functionality. The code snippet below shows that cProc provides some function calls to load or remove bitstreams and some other functionality.

```
/**
* @brief Reconfigure the vFPGA
*
* @param oid : operator ID
*/
void reconfigure(int32_t oid);
void addBitstream(std::string name, int32_t
    ↪ oid);
void removeBitstream(int32_t oid);
auto isReconfigurable() const { return fcnfg.
    ↪ en_pr; }
```

Listing 1: Code snippet from cProc class declaration

3 OS abstractions

In the last section, we discussed how communication between CPU and FPGA is done and how computation is offloaded from CPU to FPGA and then executed on FPGA. We still have one missing part in order to execute

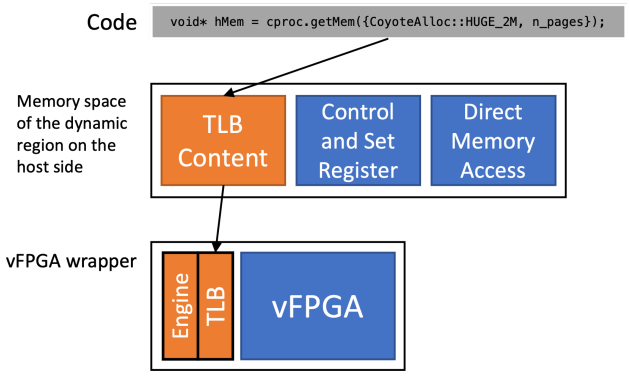

Figure 4: TLBs in Coyote

a computation, which is how the data is stored on both sides. In this section, we look at how both CPU and FPGA memory are managed.

3.1 Virtual address

Coyote leverages a similar Unified Memory strategy adopted in the CUDA programming model [11], which enables a unified shared memory pool between CPU and GPU. The programmer can allocate memory on an FPGA that is accessible to CPU code.

In Coyote, each vFPGA has its own TLBs. Something unique about Coyote is that the page size of each vFPGA can be set to either 4 KiB or 2 MiB. Different page sizes can be useful for different memory access patterns [16]. Two page size choices mean that there are two TLBs for a single vFPGA, while only one of them will be enabled. The virtual memory address is translated to the physical memory address of the FPGA by the TLBs. When there is a TLB miss, it is forwarded to the host CPU as an interrupt through a specific memory space. The software driver then either loads the TLB with a valid mapping or issues a page fault to the driver on the host.

On the host side, the memory region for each dynamic region is subdivided into three parts:

1. TLB contents and other privileged configuration values. TLB content includes how large the page size is for this vFPGA, TLB associativity, and mappings from virtual addresses to physical addresses, etc. This region is only accessible to Coyote driver.

2. Control and State Registers (CSR) on user logic. These registers are accessible to user programs, which can be used to set up DMA copies.

3. Direct access to user logic. If the user program wants to send some data directly to the user logic on the FPGA, it can write to this part. This way, data does not need to be stored first on FPGA memory and

then accessed by user logic from FPGA memory. It can be considered as a direct host-fpga streaming channel.

As shown in Figure 4, when the programmer allocates FPGA memory on the host side, the TLB contents will be filled with mappings of the host's virtual memory address to the FPGA's physical memory address. The TLB on the FPGA will also be filled with the same content by the driver on the host side.

3.2 Memory management

In Coyote, both CPU and FPGA can access each other's memory. However, handling these memory access requests with different sources and destinations can be a complicated process. Coyote introduces the read/write engine into each vFPGA to distribute memory access requests. The read/write engine and TLB together with other components such as Config Slave and queues form vFPGA's Memory Management Unit (MMU). However, unlike MMU on CPU, FPGA's MMU does not handle dynamic translation of virtual address to physical address. The mapping is actually established by the host, and the driver loads the mappings into TLBs of each vFPGA.

In Figure 5, we illustrate memory accesses by following the workflow through these numbered steps.

1. Memory access requests from either the host side or the FPGA side are first passed to Config Slave. Config Slave is to collect and distribute descriptors of all requests. Information about host requests is given directly to Config Slave, while FPGA requests need to be issued by user logic to Config Slave after receiving information such as the host's target memory address. Because the dynamic allocation of FPGA memory is done by the driver, the user logic on the FPGA has no information about the FPGA memory without communicating with the host.

2. For host requests, they are first translated into base/length descriptors by Config Slave, while FPGA requests are already in descriptor form. Config Slave and then push the descriptors into queues where the requests wait for further processing. Queues are classified according to two criteria. The first criterion is whether the queue is used to handle read or write operations; the second criterion is based on whether the request is from the host or the FPGA.

3. Read/Write engine fetches descriptors from the associated queue and prepares to issue access.

4. To access FPGA memory, the target address must be translated to the physical address for FPGA memory by TLB. While access to host memory can only come from FPGA, the target address is already the physical address given by the host, so XDMA knows where to access the data.

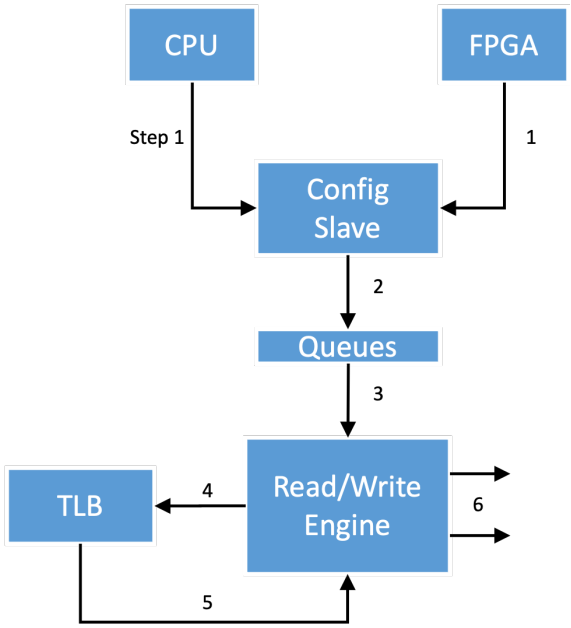

Figure 5: Memory access workflow

5. TLB returns the physical address on the hit TLB or causes an interrupt to the host when there is a TLB miss.

6. Finally, descriptors are categorized into local FPGA access and host CPU access. Then, the read/write engine forwards them to the responsible components.

The read/write engine has greatly reduced the complexity of routing memory access from both CPU and FPGA. This design also provides the ground for priority-based I/O scheduling. For example, we can schedule descriptors in queues with some priority-based algorithm so that I/O throughput can be further boosted.

4 Coyote debugging

In Coyote, computation and data are distributed to both CPU and FPGA. Debugging execution running on CPU is the same as debugging a program running purely on CPU. The difficulty lies more on the FPGA side, as there is no such debugger as GDB for FPGA. One way to debug on FPGA is to capture signals so that we can see if changes in data with respect to time are correct or not. Integrated Logic Analyzer is an IP provided by Xlinix to monitor the internal signals of a design [3].

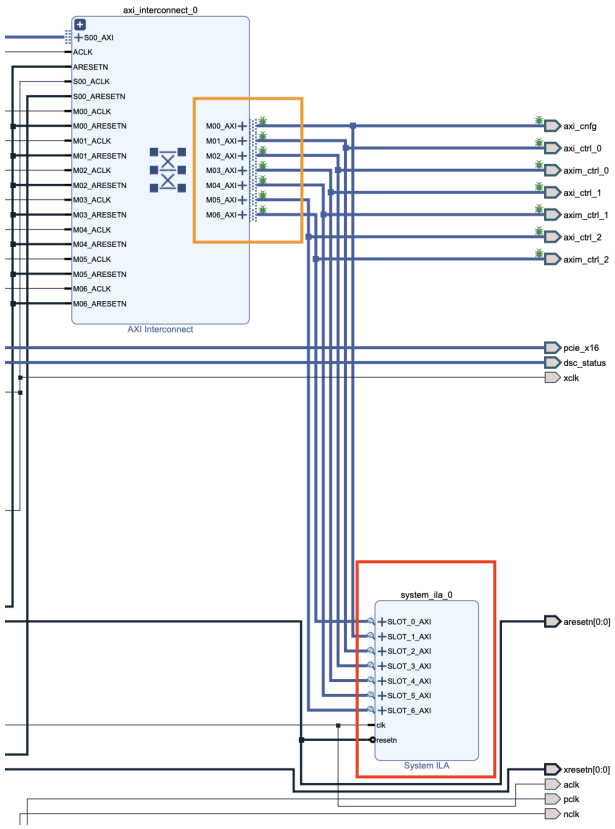

Figure 6: ILA in Block Design

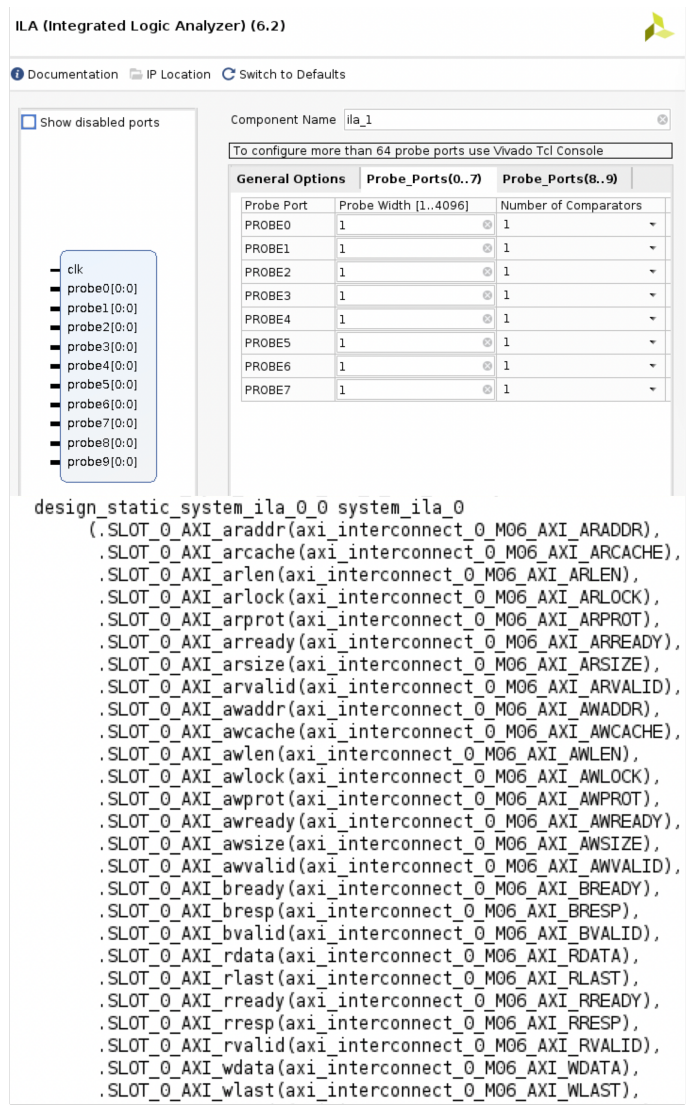

Figure 7: ILA in general way

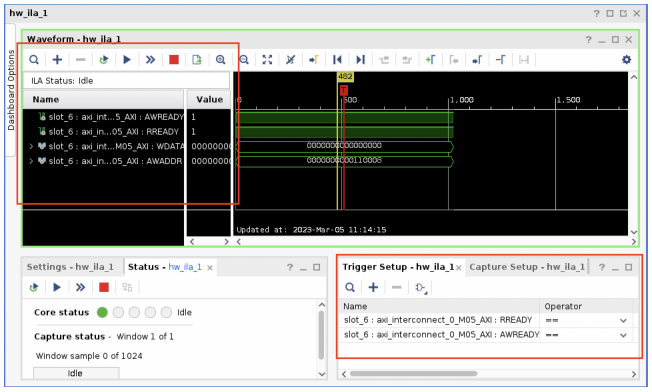

Figure 8: ILA running results

There are two ways to add ILA to the FPGA design to monitor signals. One simple way is to let Block Design automatically connect all selected signals to ILA, as shown in Figure 6. The first thing to do is to mark the desired signals with a debug flag, and then run automation. The ILA will be automatically set up and added to the design. However, this way of adding ILA is limited to Block Design. There is also a general way to add ILA, which requires more manual work. The ILA in this way must be manually configured, e.g. how many signals are there and what the widths of these signals are. Figure 7 shows an example of how to add ILA to capture any signal in the design.

After ILA is successfully added to the design, the entire project needs to be resynthesized. A bitstream shall be generated and loaded onto the FPGA board. The interface as Figure 8 should be represented. In the upper left, the signals can be further selected for tracking. In the lower right, some triggers can be set. After setup, run the ILA and wait for the signals to be picked up.

5 Evaluation

In the evaluation part, two sample testing programs provided by Coyote's author are chosen to test what we discussed in the paper. Both programs focus on data transfers. One is host-initiated data transfer and the other is FPGA-initiated data transfer. The test platform is Alveo U50 FPGA card with 8 GB of memory.

5.1 Host-initiated data transfer

In this test case, the data is sent from the host to the FPGA and then read by the host. We can see from the code snippet that the memory of the FPGA is first allocated to each vFPGA. Then we have the virtual address to the allocated memory. Now we can initiate the data transfer between the host and the FPGA.

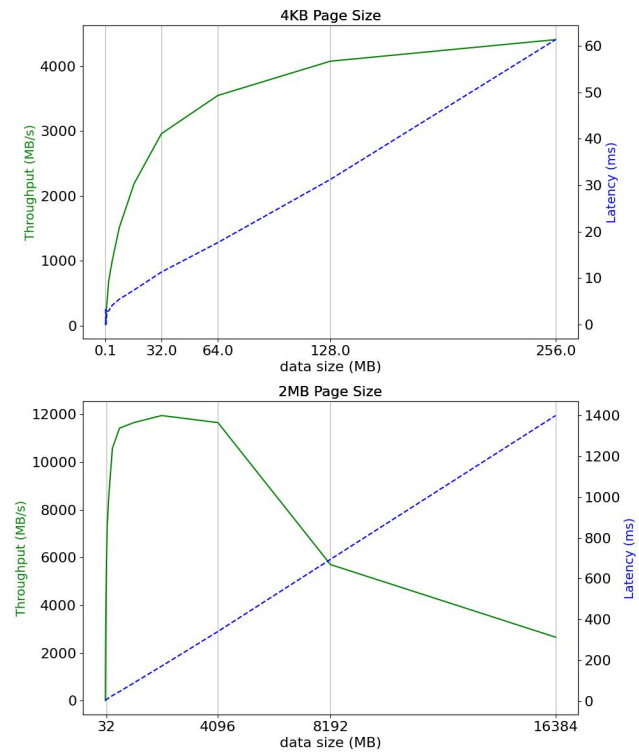

Figure 9: Results of host-initiated data transfer

```
// allocate memory on FPGA
for (int i = 0; i < n_regions; i++) {
        cproc.emplace_back(new cProc(i, getpid
            ↪ ()));
        hMem[i] = cproc[i]->getMem({huge ?
            ↪ CoyoteAlloc::HUGE_2M :
            ↪ CoyoteAlloc::REG_4K, n_pages});
}
...
...
// Transfer the data
for(int i = 0; i < n_reps; i++)
    for(int j = 0; j < n_regions; j++)
        cproc[j]->invoke({CoyoteOper::TRANSFER,
            ↪  hMem[j], hMem[j], curr_size,
            ↪ curr_size, false, false});
```

Listing 2: Code snippet of host-initiated data transfer

The operator *CoyoteOper::TRANSFER* means a data transfer loopback. The data will have a round trip be-

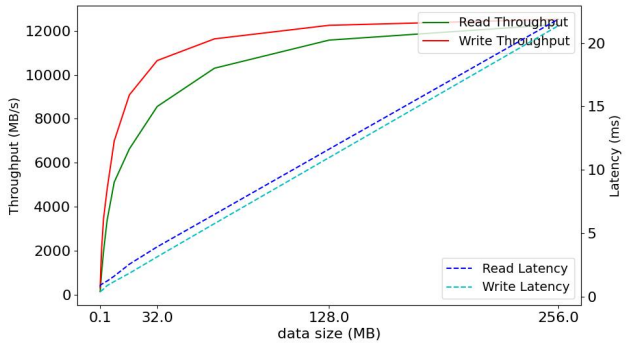

Figure 10: Results of FPGA-initiated data transfer

tween CPU and FPGA. As discussed in the previous section, the memory request in this case will be given directly to Config Slave and distributed to the read or write engine.

Test results for throughput and latency with different page sizes are given in Figure 9. We can clearly see that it has higher throughput when the page size is 2 MB. Since this data transfer is initiated by the host, the target FPGA memory addresses need to be translated into physical addresses. Therefore, in this case, a larger page size can reduce the miss rate, thus providing better performance. The throughput drops dramatically when the payload size reaches 8 GB. That's because the FPGA card's memory for testing has only 8 GB of on-board memory.

5.2 FPGA-initiated data transfer

The data transfer initiated by the FPGA means that the FPGA will attempt to access memory on the host via DMA. A DMA requires information such as the source data address and data size. As shown in the following code snippet, all the necessary information is first given to the user logic on the FPGA via the CSR memory region by setting the CSR to cProc. With the information on the memory request, user logic can hand over the memory request to Config Slave for further processing.

If we compare code snippets from both examples, we should note that they validate what we discussed in section 3.2. Host-initiated data transfer invokes the data transfer directly with the relevant information, while FPGA-initiated data transfer needs to first give the relevant information to FPGA, then the data transfer is invoked by FPGA.

The results of the data transfer initiated by FPGA are shown in Figure 10. It is shown that both throughput and latency increase as the size of the data is transferred. However, the maximum throughput on this test platform is around 12 GB/s. Moreover, write operations have higher throughput and lower latency than read operations, which can be a factor to consider when designing the priority-based I/O scheduler.

```
// Run Throughput
auto benchmark_run = [](cProc cproc, const
    ↪ void* hMem, const BenchOper oper) {
    // Set params
    cproc.setCSR(reinterpret_cast<uint64_t>(
        ↪ hMem), static_cast<uint32_t>(
        ↪ BenchRegs::VADDR_REG));
    cproc.setCSR(curr_size, static_cast<
        ↪ uint32_t>(BenchRegs::LEN_REG));
    cproc.setCSR(cproc.getCpid(), static_cast<
        ↪ uint32_t>(BenchRegs::PID_REG));
    cproc.setCSR(n_reps, static_cast<uint32_t>
        ↪ (BenchRegs::N_REPS_REG));

    uint64_t n_beats = n_reps * ((curr_size +
        ↪ 64 - 1) / 64);
    cproc.setCSR(n_beats, static_cast<uint32_t
        ↪ >(BenchRegs::N_BEATS_REG));

    // Fire
    cproc.setCSR(static_cast<uint64_t>(oper),
        ↪ static_cast<uint32_t>(BenchRegs::
        ↪ CTRL_REG));

    while(cproc.getCSR(static_cast<uint32_t>(
        ↪ BenchRegs::DONE_REG)) < n_reps) ;
    return (double)(cproc.getCSR(static_cast<
        ↪ uint32_t(BenchRegs::TIMER_REG))) *
        ↪ clkNs;
    };
...
...
// through/latency test
for(int j = 0; j < nBenchRuns; j++) {
    time_bench_rd.emplace_back(benchmark_run(
        ↪ cproc, hMem, BenchOper::START_RD));
    time_bench_wr.emplace_back(benchmark_run(
        ↪ cproc, hMem, BenchOper::START_WR));
}
```

Listing 3: Code snippet of FPGA-initiated data transfer

6 Conclusion

Coyote provides a way to solve the problem that programmers find difficult to program FPGA by bringing OS abstraction to FPGA. Static and dynamic region allows vFPGA so that we can have process abstraction to program FPGA. Virtual memory provides a unified memory address space. It works with read/write engines on FPGA for data transfer. A potential I/O scheduler can be given to issue memory access descriptors in a priority-based approach to further increase throughput and reduce latency.

This paper presents a study of Coyote memory system. Much remains to be studied in other parts of the system. For example, how TLB misses are actually handled by the driver and how the dynamic region in the PCIe address space of a vFPGA is managed. However, this paper should provide sufficient knowledge to provide an overview of how Coyote works.

References

[1] Advanced Micro Devices, Inc. *AMBA AXI4 Interface Protocol*. https://www.xilinx.com/products/intellectual-property/axi.html#overview, February 2023.

[2] Advanced Micro Devices, Inc. *DMA for PCI Express (PCIe) Subsystem*. https://www.xilinx.com/products/intellectual-property/pcie-dma.html, February 2023.

[3] Advanced Micro Devices, Inc. *Integrated Logic Analyzer (ILA)*. https://www.xilinx.com/products/intellectual-property/ila.html, March 2023.

[4] Alibaba Cloud Services. *Compute optimized type family for FPGA*. https://www.alibabacloud.com/help/doc-detail/108504.htm, February 2023.

[5] Amazon Web Services. *Amazon EC2 F1 Instances*. https://aws.amazon.com/ec2/instance-types/f1/, February 2023.

[6] P.M. Athanas and H.F. Silverman. "Processor reconfiguration through instruction-set metamorphosis". In: *Computer* 26.3 (1993), pp. 11–18. DOI: 10.1109/2.204677.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[8] Fasahat Hussain et al. "Exploring FPGA Logic Block Architecture for Reduced Configuration Memory". In: *Advances in Electrical and Computer Engineering* 22.3 (2022), pp. 15–24.

[9] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. "Do OS abstractions make sense on FPGAs?" In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 2020, pp. 991–1010.

[10] Bojie Li et al. "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 1–14.

[11] Wenqiang Li et al. "An Evaluation of Unified Memory Technology on NVIDIA GPUs". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 1092–1098. DOI: 10.1109/CCGrid.2015.105.

[12] Oracle Corporation. *PCI Address Domain*. https://docs.oracle.com/cd/E19253-01/816-4854/hwovr-25/index.html, February 2023.

[13] Philippos Papaphilippou, Jiuxi Meng, and Wayne Luk. "High-Performance FPGA Network Switch Architecture". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 76–85. ISBN: 9781450370998. DOI: 10.1145/3373087.3375299. URL: https://doi.org/10.1145/3373087.3375299.

[14] Andrew Putnam et al. "A reconfigurable fabric for accelerating large-scale datacenter services". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.

[15] Vishal Shrivastav. "Fast, scalable, and programmable packet scheduler in hardware". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 367–379.

[16] Madhusudhan Talluri et al. "Tradeoffs in Supporting Two Page Sizes". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. Queensland, Australia: Association for Computing Machinery, 1992, pp. 415–424. ISBN: 0897915097. DOI: 10.1145/139669.140406. URL: https://doi.org/10.1145/139669.140406.