Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment

by

Wei Wang

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements for the degree of Masters of Applied Science, Electrical and Computer Engineering

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

In this thesis we present pvFPGA, the first system design solution for virtualizing an FPGA-based hardware accelerator on the x86 platform. The accelerator design on the FPGA can be used for accelerating various applications, regardless of the application computation latencies. Our design adopts the Xen virtual machine monitor (VMM) to build a paravirtualized environment, and a Xilinx Virtex-6 as an FPGA accelerator. The accelerator communicates with the x86 server via PCI Express (PCIe). In comparison to the current GPU virtualization solutions, which primarily intercept and redirect API calls to the hosted or privileged domain's user space, pvFPGA virtualizes an FPGA accelerator directly at the lower device driver layer. This gives rise to higher efficiency and lower overhead. In pvFPGA, each unprivileged domain allocates a shared data pool for both user-kernel and inter-domain data transfer. In addition, we propose the coprovisor, a new component that enables multiple domains to simultaneously access an FPGA accelerator. The experimental results have shown that 1) pvFPGA achieves close-to-zero overhead compared to accessing the FPGA accelerator without the VMM layer, 2) the FPGA accelerator is successfully shared by multiple domains, 3) distributing different maximum data transfer bandwidths to different domains can be achieved by regulating the size of the shared data pool at the split driver loading time, 4) request turnaround time is improved through DMA(Direct Memory Access) context switches implemented by the coprovisor.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Miodrag Bolic, for giving me the opportunity to work on a topic of great interest to me. His encouragement, support and excellent guidance helped me understand and develop this project. I would also like to thank PhD student Jonathan Parri for his great help with the project, as well as the members of Computer Architecture Research Group (CARG) I had the pleasure of working with. As well, a big thank you goes to my family for their support and encouragement during the work.

Finally, I wish to thank CMC Microsystems for their assistance with the experimental platform, and NSERC for the financial support they provided for this research.

Contents

List of Figures ..... Ⅴ

List of Tables ..... Ⅶ

List of Abbreviations ..... Ⅷ

1 Introduction 1

1.1 Introduction of Virtualization 1

1.2 Motivation 2

1.3 Challenges and Contributions 3

1.4 Thesis Organization 5

2 Background 7

2.1 FPGAs 7

2.1.1 Basic Concepts 7

2.1.2 FPGA Programming and Design Flow 8

2.1.3 Partial Reconfiguration 11

2.2 FPGA-based Hardware Acceleration 12

2.3 An Overview of the x86 Hardware Architecture 14

2.4 x86 Virtualization 16

2.5 Types of VMM 18

2.6 Introduction to the Xen VMM 19

2.6.1 Priority Levels in the Xen VMM 20

2.6.2 Domains 20

2.6.3 Memory Management 21

2.6.4 Mechanisms 23

2.6.5 Xen Scheduler 24

2.7 User Space and Kernel Space 25

2.8 An Overview of PCI Express 27

2.8.1 PCI-based Systems 27

2.8.2 PCI Express 29

2.9 Direct Memory Access 30

3 Basic Framework Design 31

3.1 FPGA Hardware Accelerator Design 32

3.1.1 Design with One DMA Channel 32

3.1.2 Design with Two DMA Channels 34
3.2 Design using the Pipeline Technique 35
3.3 Device Driver Design for the FPGA Accelerator 39
3.4 FPGA Virtualization 40
3.4.1 Data Transfer 40
3.4.2 Construction of the Data Transfer Channel 44
3.4.3 Coprovisor 45

4 pvFPGA Amelioration 48
4.1 Analysis 48
4.2 FPGA Accelerator Design Modification 50
4.2.1 App Controller 50
4.2.2 Accelerator Status Word 50
4.3 Coprovisor Design Modification 53

5 Implementation and Evaluation 56
5.1 Experiments Introduction 56
5.2 Accelerator Design Evaluation 57
5.3 Virtualization Overhead Evaluation 59
5.4 Verification with Fast Fourier Transform 63
5.5 Coprovisor Evaluation 66
5.5.1 Contention Analysis 68
5.6 DMA Context Switching Evaluation 72

6 Related Work 74
6.1 FPGA Virtualization 74
6.1.1 Design Review 74
6.1.2 Comparison with pvFPGA 75
6.2 Inter-domain Network Virtualization 75
6.2.1 Design Review 75
6.2.2 Comparison with pvFPGA 78
6.3 GPU Virtualization 78
6.3.1 Design Review 78
6.3.2 Comparison with pvFPGA 80

7 Conclusion and Future Work 82
7.1 Conclusion 82
7.2 Future Work 83

Appendix A 85

Bibliography 88

List of Figures

Figure 1. State-of-the-art GPU virtualization and FPGA virtualization 3
Figure 2. A Virtex-6 configurable logic block [24] 8
Figure 3. An example of FPGA design flow 9
Figure 4. An example of partial reconfiguration 11
Figure 5. An example of using an FPGA hardware accelerator 13
Figure 6. Time for executing compute intensive algorithms 13
Figure 7. An overview of x86 hardware platform 15
Figure 8. A bare metal VMM 18
Figure 9. A hosted VMM 19
Figure 10. An overview of Xen-based paravirtualized environment 21
Figure 11. A pseudo-physical memory model 22
Figure 12. System calls in a paravirtualized system 26
Figure 13. An example of PCI-based system 28
Figure 14. The layout of pvFPGA design 31
Figure 15. Design of an FPGA accelerator with one DMA channel 33
Figure 16. Design of an FPGA accelerator with two DMA channels 34
Figure 17. Examples of non-RISC pipelined operations 36
Figure 18. Delayed operations in the pipeline technique 38
Figure 19. Simultaneous implementation of DMA read and write operations 38
Figure 20. DMA buffer descriptors 40
Figure 21. Inter-domain communication design of pvFPGA 41
Figure 22. Dataflow in pvFPGA 43
Figure 23. The coprovisor design of pvFPGA 46

Figure 24. Analysis of request turnaround time 49
Figure 25. Design of a request control block 52
Figure 26. The improved coprovisor design 53
Figure 27. Request state transition diagram 54
Figure 28. DMA read channel state transition diagram 54
Figure 29. Evaluation of the non-pipelined and pipelined FPGA design 58
Figure 30. Pipeline of the loopback application 59
Figure 31. Time for executing a loopback application on the FPGA accelerator 61
Figure 32. Overhead evaluation with a loopback application 62
Figure 33. Verification with an FFT application 64
Figure 34. Overhead evaluation with an FFT application 65
Figure 35. Evaluation of the coprovisor 67
Figure 36. Analysis of four DomUs contending for access to the shared FPGA accelerator 69
Figure 37. An accelerator emulating design for verification purposes 72
Figure 38. Request turnaround time comparison between the basic design and improved design 73
Figure 39. An example of ARP cache content generated from Dom0 77

List of Tables

Table 1. Design of the ASW..... 51
Table 2. Bits in the ASW..... 51
Table 3. Description of a request control block 52
Table 4. Experimental platform 56
Table 5. Remaining data size of each DomU at the end Phase1..... 70
Table 6. Remaining data size of each DomU at the end Phase2..... 71
Table 7. An example of XenLoop table strored in a DomU 77
Table 8. Comparison of pvFPGA with the current GPU virtualization solutions..... 81

List of Abbreviations

ALU	Arithmetic and Logic Unit
API	Application Programming Interfaces
APIC	Advanced Programmable Interrupt Controller
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
ASW	Accelerator Status Word
BAR	Base Address Register
BIOS	Basic Input/Output System
CLB	Configurable Logic Block
CML	Communication Latency
CPU	Central Processing Unit
CPL	Computation Latency
CUDA	Compute Unified Device Architecture
Dom0	Domain 0
DMA	Direct Memory Access
DomU	Unprivileged Domain
DW	Double Word
FCFS	First-Come, First-Served
FFT	Fast Fourier Transform
FIFO	First-In-First-Out
FPGA	Field-Programmable gate array
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language

HPC	High Performance Computing
HVM	Hardware-assisted Virtual Machine
IC	1) Integrated Circuit; 2) Interrupt Controller
ICH	Input/Output Controller Hub
IDT	Interrupt Descriptor Table
IF	Interrupt Flag
I/O	Input and Output
IOMMU	Input/Output Memory Management Unit
IP	1) Intellectual Property; 2) Internet Protocol
IPC	Inter-Process Communication
ISA	Industry Standard Architecture
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LUT	Look-up Table
MAC	Media Access Control
MCH	Memory Controller Hub
MMU	Memory Management Unit
MSI	Message Signaled Interrupt
MUX	Multiplexers
NUMA	Non-Uniform Memory Access
OS	Operating System
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PIO	Programmed Input/Output
QEMU	Quick EMUlator
QoS	Qualities of Service
QPI	QuickPath Interconnect

RCB	Request Control Block
RCU	Read-Copy-Update
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Level
SMP	Symmetric Multiprocessing
TCP	Transmission Control Protocol
TLB	Translation Look-aside Buffer
UDP	User Datagram Protocol
VCM	Virtual Coprocessor Monitor
vCPU	Virtual CPU
VIP	Very Important Person
VMM	Virtual Machine Monitor

Chapter 1

Introduction

1.1 Introduction of Virtualization

Cloud computing, which refers to both the applications delivered as services over the Internet and the hardware along with systems software in the data centers that provide those services [1], has taken center stage in information technology in recent years. As a key component of cloud computing, virtualization technology has gained tremendous attention. Virtualization technology can be categorized as application-level oriented virtualization and system-level oriented virtualization.

A classic example of application-level virtualization is Java virtual machine (JVM) [2]. JVM is a stack-based virtual machine that implements Java bycodes in memory stacks, thereby making no assumptions about the processor architectures (i.e. registers). Thus, it offers a virtualization layer for the high level Java code, and this layer gives Java language great portability, as noted by its slogan created by Sun Microsystems, “Write once, Run anywhere”.

System-level virtualization refers the technology of virtualizing an entire system. With system-level virtualization technology, the underlying hardware resources can be shared by multiple virtual machines or domains with each running its own operating system (OS). This gives rise to higher hardware utilization and lower power consumption. The key component in system-level virtualization is the Virtual Machine Monitor (VMM), also referred to as a hypervisor. A VMM is responsible for isolating each running instance of an operating system from the physical machine. The VMM translates or emulates special instructions of a guest OS. A VMM itself is a complex piece of software, which

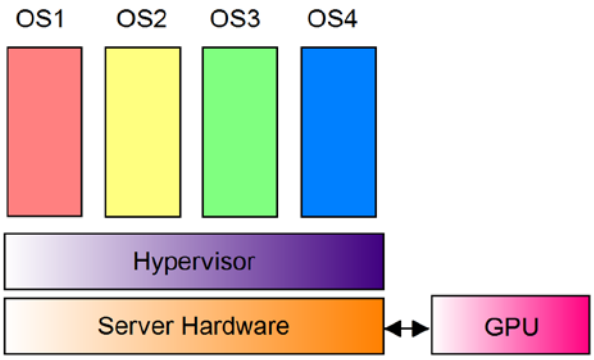
typically runs with the highest privilege level (higher than a guest OS), so it is vital to ensure a VMM is as bug-free as possible.

1.2 Motivation

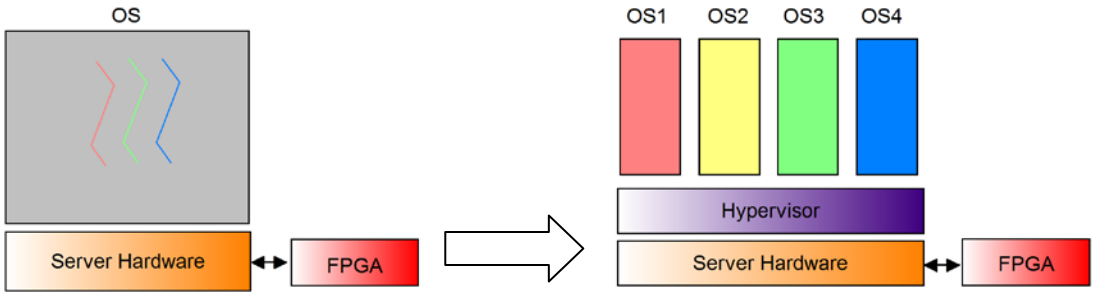
Graphics processing unit (GPU) and Field-Programmable gate array (FPGA) based hardware accelerators have been gaining popularity in the server industry. A hardware accelerator, which plays the role of a coprocessor of a central processing unit (CPU), speeds up computationally-intensive parts of an application. It is not uncommon for a hardware accelerator to be used for accelerating video/image processing [3, 4], signal processing [5] and various mathematical calculations [6, 7]. Successfully and efficiently adding hardware accelerators to virtualized servers will bring cloud clients apparent speedup for a wide range of applications.

GPUs are inexpensive, and they are typically programmed using high level languages and application programming interfaces (APIs) which abstract away hardware details [8]. Despite many challenges of making GPUs a shared resource in a virtualized environment, many papers [9-14] have succeeded in virtualizing GPUs by a VMM (Figure 1(a)). FPGAs have been found to outperform GPUs in many specific applications [8, 15]. The ability to perform partial run-time reconfiguration [16] is an important distinguishing feature of FPGAs. Some effort has already been put into FPGA virtualization [17- 21], but the research still stays at multitasking level on a single OS.

The main objective of this thesis is to move a step forward in the virtualization of an FPGA for its use as a hardware accelerator by a VMM on an x86 server (Figure 1(b)). We focus on designing a scheme in which a user process from a guest domain can access an FPGA accelerator with low overhead. Additionally, our proposed solution will likely have the capability to efficiently multiplex multiple guest domains to simultaneously request access to a shared FPGA accelerator.



a) Virtualizing a GPU by a hypervisor



b) Moving the state-of-the-art FPGA virtualization toward the hypervisor layer

Figure 1. State-of-the-art GPU virtualization and FPGA virtualization

1.3 Challenges and Contributions

As mentioned in Section 1.2, many works have succeeded in virtualizing GPUs by a VMM. The architecture of a GPU is designed by a hardware vendor, and the GPU specification is held by them privately. The only way to access the GPU is to use the vendor supplied user library calls (e.g. CUDA library). Due to the unavailability of the underlying GPU driver, these virtualization solutions need to redirect the requests from guest OSes to the privileged or host OS's user space, where they can invoke a library call to access the GPU. We need to come up with an accelerator design for FPGAs, and we also need to create the device driver for the FPGA, so that the virtualization work can be directly implemented at the device driver layer.

To the best our knowledge, no solutions have yet been proposed for a VMM

incorporating FPGA acceleration for their virtual machines. We propose the first solution for sharing an FPGA accelerator in a virtualized environment. Paravirtualization is well known for its high Input and Output (I/O) performance compared to other types of virtualization. We adopt the Xen VMM to build a paravirtualized environment where an FPGA accelerator is accessed simultaneously by multiple domains. We term the tailored-made solution pvFPGA.

Hardware accelerators are commonly used for accelerating computationally intensive applications, which typically require large amounts of data for the calculations. To tailor pvFPGA to a high efficient solution, four factors relating to data transfer need to be considered:

Server-Accelerator Data Transfer: Large amounts of data need to be efficiently transferred between an FPGA accelerator and an x86 server so that apparent speedup can be achieved.

User-Kernel Data Transfer: A user application needs to transfer large amounts of data to the driver which resides in kernel space. This data transfer should be as fast as possible.

Inter-domain Data Transfer: A guest domain needs to transfer large amounts of data to a privileged domain which has the priority to directly access an FPGA accelerator. The inter-domain data transfer is an overhead introduced by the virtualization layer, and must be as minimal as possible.

Flexible Data Transfer: A user application in a guest domain should be able to specify variable sizes of data that it needs to transfer to an FPGA accelerator for the calculation.

In addition, pvFPGA should be capable of enabling multiple domains to access an FPGA accelerator simultaneously. Service is the theme of cloud computing. Different qualities of service (QoS) are offered to cloud clients, according to the rent they pay the cloud vendor. Similarly, pvFPGA should be able to supply guest domains with different

maximum data transfer bandwidths, so that VIP (Very Important Person) clients can finish their acceleration requests faster than other clients. To offer flexible choices for clients, pvFPGA supports multiple applications running on an FPGA accelerator. The applications can be dynamically selected by a guest domain during runtime.

Overall, the main contributions of this thesis are as follows:

1) the thesis proposes the first system design solution in the virtualization of an FPGA for its use as a hardware accelerator by a VMM;
2) the overhead caused by the virtualization layer is close to zero;
3) we propose a component called a coprovisor, which successfully multiplexes multiple domains to access a shared FPGA accelerator simultaneously;
4) VIP guest domains have higher maximum data transfer bandwidths than ordinary guest domains in our solution;
5) We further improve the basic design with a capability of implementing DMA context switches, which is intended to reduce the request turnaround time.

Parts of the project are published in [66].

1.4 Thesis Organization

This section provides a brief outline of the content of the individual chapters. The thesis is organized as follows:

Chapter 2 presents background information about concepts related to this project. It includes introductions to FPGAs, FPGA-based hardware acceleration, x86 platforms, virtualization technology, the Xen VMM, PCI Express and DMA etc.

Chapter 3 elaborates on the hardware/software co-design of pvFPGA. We propose two types of FPGA accelerator design, and a custom virtualization solution for FPGA accelerators. In addition, several cases of pipelined operations are detailed in this chapter.

Chapter 4 analyses the limitations of the basic design proposed in Chapter 3. The improved design is intended to reduce the scheduling delay with DMA context switches,

thereby reducing the request turnaround time.

Chapter 5 presents the implementation and evaluation of the pvFPGA design. The performance of the two FPGA accelerator designs is compared, and the overhead caused by the virtualization layer is evaluated. We verify the pvFPGA design with a Fast Fourier Transform (FFT) benchmark. We then evaluate pvFPGA with multiple guest domains accessing the shared FPGA accelerator simultaneously. The last section of this chapter evaluates the improvement brought by the DMA context switches proposed in Chapter 4.

Chapter 6 presents related work. We research the state-of-the-art FPGA virtualization for a multitasking OS and the latest inter-domain network virtualization, as well as related GPU virtualization solutions.

Chapter 7 summarizes the current state of our work, and proposes potential future research.

Chapter 2

Background

2.1 FPGAs

2.1.1 Basic Concepts

Field programmable gate arrays (FPGA) are digital integrated circuits (IC) that contain configurable (programmable) logic blocks (CLB) along with configurable interconnects between the blocks [22]. Unlike Application Specific Integrated Circuits (ASIC), which perform a single specific function for the lifetime of the chip, an FPGA can be reprogrammed to perform a different function in microseconds [23]. Therefore, FPGAs are commonly used for ASIC prototyping.

CLBs are the main logic resources for implementing both sequential and combinatorial circuits [24]. Figure 2 shows a Virtex-6 CLB composed of two slices. Each slice contains four look-up tables (LUT), eight storage elements and several multiplexers (MUX). An LUT can be configured into a basic logic unit such as an “and” gate. A storage element can be configured into either an edge-triggered D-type flip-flop or a level-sensitive latch. The MUXs enable a flexible implementation of logic functions within an FPGA fabric. Thus, a slice is able to offer logic, arithmetic and storage functions. Modern FPGAs also have dedicated blocks (e.g. embedded multipliers) that can work at higher operating frequencies, and take up less area than those made up of CLBs.

Most of today’s FPGAs have abundant resources, such as the 37680 slices in a Virtex-6 LX240T FPGA. In most cases, these resources are sufficient to implement a system to solve complicated mathematical problems. Thus, FPGAs have become

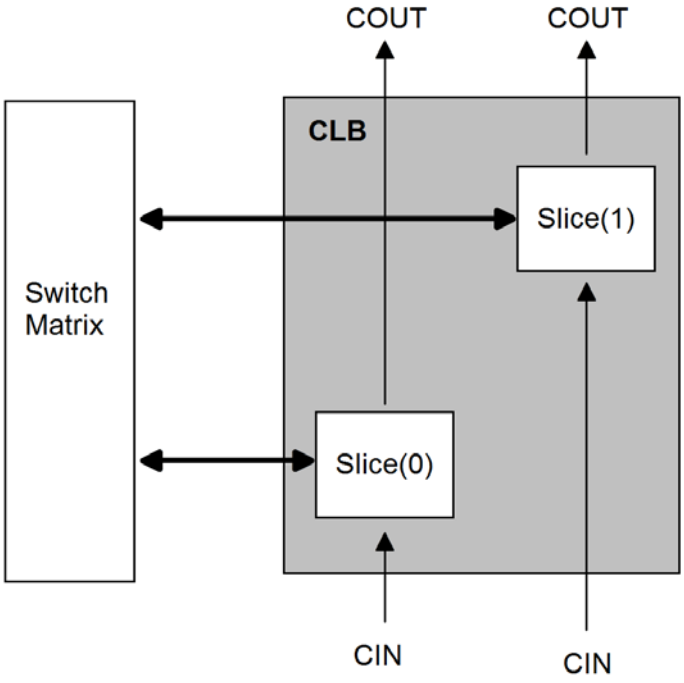


Figure 2. A Virtex-6 configurable logic block [24]

commonplace in various fields such as medical field, aerospace and defense.

2.1.2 FPGA Programming and Design Flow

An FPGA is usually programmed with a hardware description language (HDL), such as VHDL [25] or Verilog HDL [26]. An HDL allows designers to model the concurrency of processes found in hardware elements [27]. Unlike programming a microprocessor using a sequential programming language (e.g. C), programming with an HDL requires that programmers keep hardware digital circuits in mind, because elements or modules described in hardware used to happen in parallel, and most of them also work in a timed sequential order.

Figure 3 illustrates the design flow from an HDL code to the final bitstream file which can be downloaded to the FPGA chip. The steps are as follows:

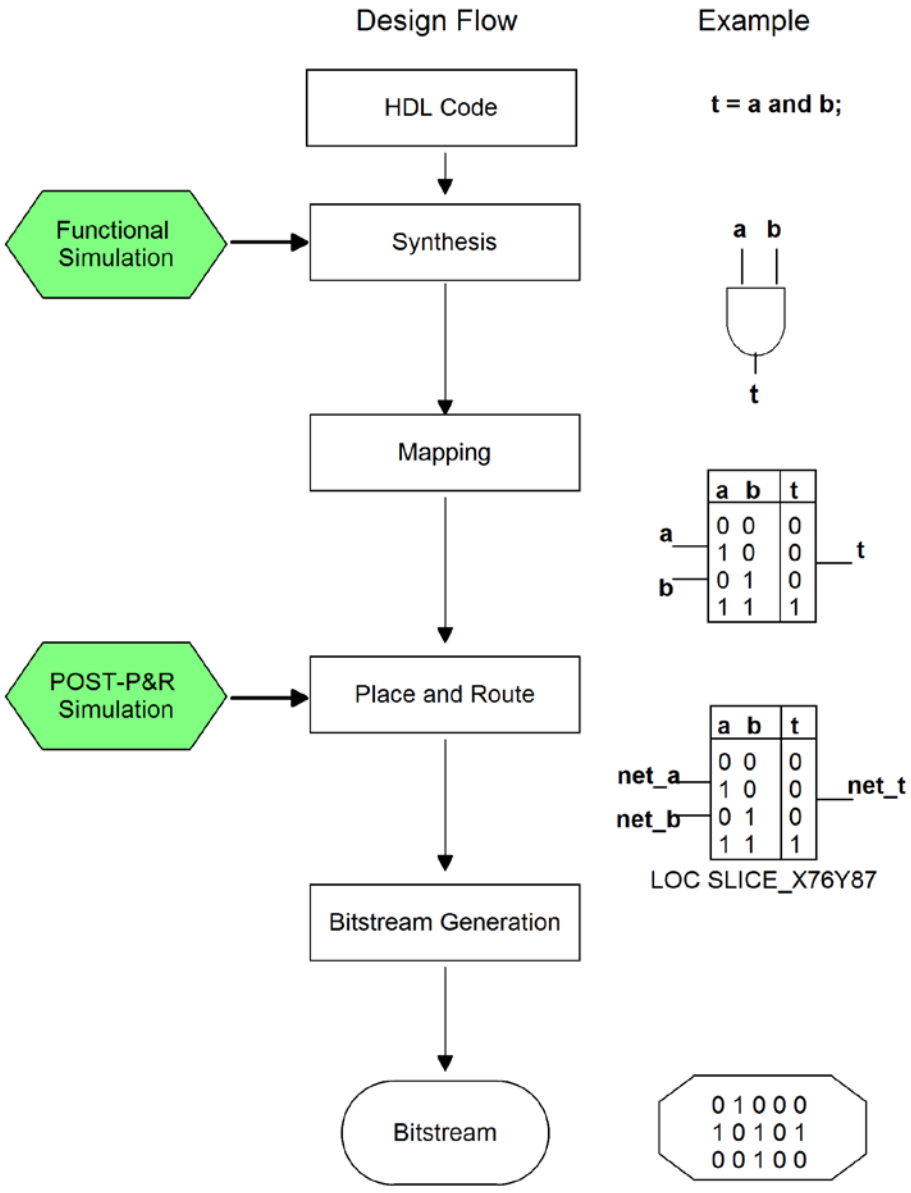


Figure 3. An example of FPGA design flow

1) Synthesis

A synthesis tool (e.g. ISE for Xilinx FPGAs, QuartusII for Altera FPGAs) checks the syntax of the entire HDL code, and translates the code at the register-transfer level (RTL). As the example shows, the HDL sentence describing an “and” function is translated into an “and” gate. A netlist which describes the connectivity of elements in the design is generated in this step.

2) Mapping

This step maps the RTL netlist into the specified FPGA hardware architecture. The "and" gate in the example is mapped into a LUT in a slice.

3) Place and Route

In this step, the mapped netlist is placed into the FPGA's fabric, and then these placed components are connected. The FPGA designer needs to specify an optimization strategy (e.g. timing, speed or balanced), which influences the position that components are placed in the FPGA's fabric.

4) Bitstream Generation

This step turns the design into a bitstream file that can be directly downloaded into the targeted FPGA.

Typically, there are two types of simulation in an FPGA design flow: functional simulation, also known as behavior simulation, and post-place-and-route simulation. Functional simulation is a very important step in the FPGA design flow, as it verifies the logic result of the designed circuit. As shown in Figure 2, a functional simulation can be implemented before the synthesis. In many cases, the mapping and place and route procedure require a couple of hours to finish, so a mistake found through the functional simulation can save a designer a lot of time.

Post-place-and-route simulation, as the name indicates, is implemented after the place and route procedure. While functional simulation is conducted on an ideal case without considering the circuit and routing delays, the post-place-and-route simulation needs to add some delay parameters to the test bench file, to simulate a real executing environment of the design on an FPGA. For example, consider the "and" gate in Figure 2. The output "c" should be 1 when input "a" and "b" are both 1. If there is a substantial delay inputting signal "b", the result output "c" could be 0, because the input "b" reaches the gate at the following cycle due to the routing delay. However, in practice, many designers choose to skip the post-place-and-route simulation by verifying the design directly on the real FPGA

board, as the bitstream file generation procedure usually needs only a couple of minutes to finish. Both the functional and post-place-and-route simulations can be implemented in Modelsim [28].

2.1.3 Partial Reconfiguration

Partial reconfiguration on FPGAs is a recently new technique. Partial reconfiguration is the ability to dynamically modify blocks of logic by downloading partial bitstream files while the remaining logic continues to operate without interruption [29]. Partial reconfiguration provides FPGAs a mechanism for hardware multitasking.

Figure 4 shows an example of partial reconfiguration. The FPGA design is comprised of four function modules. Modules A, B and D are designed together as one static module, and module C and its backup modules C1, C2 and C3 are designed as dynamic modules separately. Thus, one static bitstream file and four partial bitstream files are generated in this example. During runtime, the Function C module can be replaced with other dynamic modules by downloading only the related partial bitstream file, while the static parts keep running.

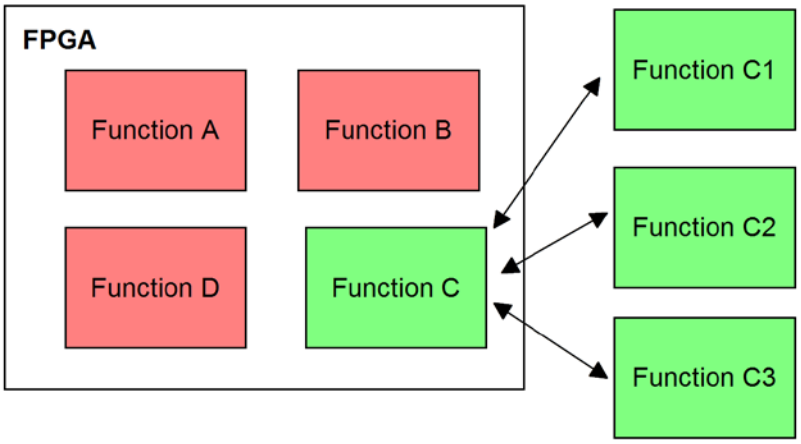


Figure 4. An example of partial reconfiguration

2.2 FPGA-based Hardware Acceleration

In general, hardware acceleration refers to the use of extra hardware acting as a coprocessor. The coprocessor assists a general purpose CPU by speeding up the calculation of complex algorithms. Figure 5 illustrates a representative example of using an FPGA as a hardware accelerator. The FPGA accelerator is typically loaded with specifically designed logic which takes advantage of parallelization and/or pipeline to speed up the calculation of complicated algorithms.

Normally, the total time for executing a computationally-intensive application on a CPU can be divided into three separate times: preprocessing time (T1), computation time (T2), and post-processing time (T3), as shown in Figure 6(a). When the computation kernel is ported to an FPGA hardware accelerator, an extra overhead time, or communication latency, must be introduced for sending data and receiving results. Figure 6(b) presents an example that an FPGA hardware accelerator outperforms a CPU with eight times speedup in terms of algorithm computation latency. The communication latency can be improved by using a higher bandwidth communication channel, for example, an 8-lane PCI Express (PCIe) interface will abate the communication latency to 1/8 of that of using a 1-lane PCIe interface. The preprocessing and post-processing procedures are, in effect, the sequential parts of a program executed on a CPU. The communication latency can be viewed as a fixed value with the determined design on the FPGA accelerator. According to Amdahl's Law, the overall speedup should be lower than the eight-fold algorithm computation speedup gained by an FPGA accelerator. This can be seen in Figure 6, where the overall speedup is a little over 1.2-fold.

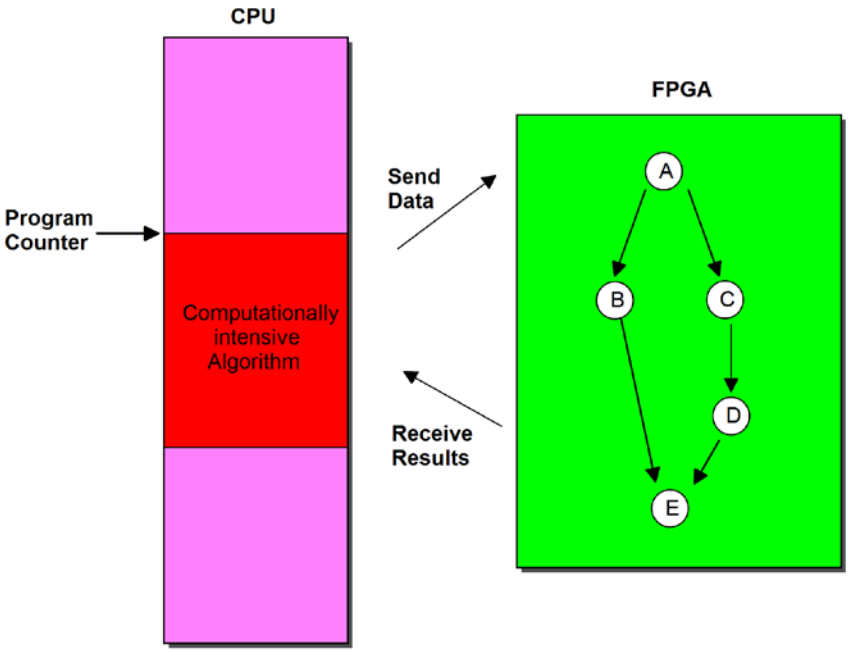


Figure 5. An example of using an FPGA hardware accelerator

a) CPU only

b) Algorithms computed on an FPGA accelerator with 8 times speedup

Figure 6. Time for executing compute intensive algorithms

However, in order to achieve speedup, two elements should be considered: CPL (Computation Latency) and CML (Communication Latency). The following formula must be satisfied:

$$CPL_{CPU} - CPL_{FPGA} > CML$$

Otherwise, no speedup can be obtained by using the hardware accelerator.

2.3 An Overview of the x86 Hardware Architecture

The first 16-bit Intel 8086 microprocessor, which was released in 1978, is the forerunner of all x86-architecture-based processors. An 8086 processor has only a 20-bit address bus, giving it exactly 1MB of address space. It works in real mode, which offers no support for memory protection. Protected mode was introduced to the x86 architecture with the release of the 80286 processors, and this mode was later enhanced in the 80386 processors.

In protected mode there are four priority levels, referred to as rings. Ring 0 is set as the highest priority, and is usually reserved for the kernel of an OS. Applications typically reside in ring 3, which is set as the lowest priority. The use of rings restricts user applications from accessing data, call gates or executing privileged instructions [30]. Most of today's x86 processors boot in real mode, and enter protected mode after the bootloader hands off control to the OS.

Figure 7 shows an overview of a classic x86 hardware platform, which consists mainly of CPUs, a northbridge and a southbridge.

1) CPU

Most modern computer systems contain multiple CPUs, and each CPU has hardware support for multithreading. For example, a Xeon W3670 processor has 6 CPU cores, and supports 12 threads in total. A CPU carries out the instructions of a computer program, and as a minimum, consists of a data path and a control unit. The data path includes registers and an arithmetic and logic unit (ALU). The hardwired or microprogrammed control unit interprets instructions and effects register transfers.

Memory Management Unit (MMU)

An MMU in a CPU is responsible for handling all memory accesses from the CPU. It essentially serves two purposes, 1) translating virtual addresses to physical addresses using page structures maintained by an OS, and 2) enforcing memory protections.

2) NorthBridge (aka Memory Controller Hub(MCH))

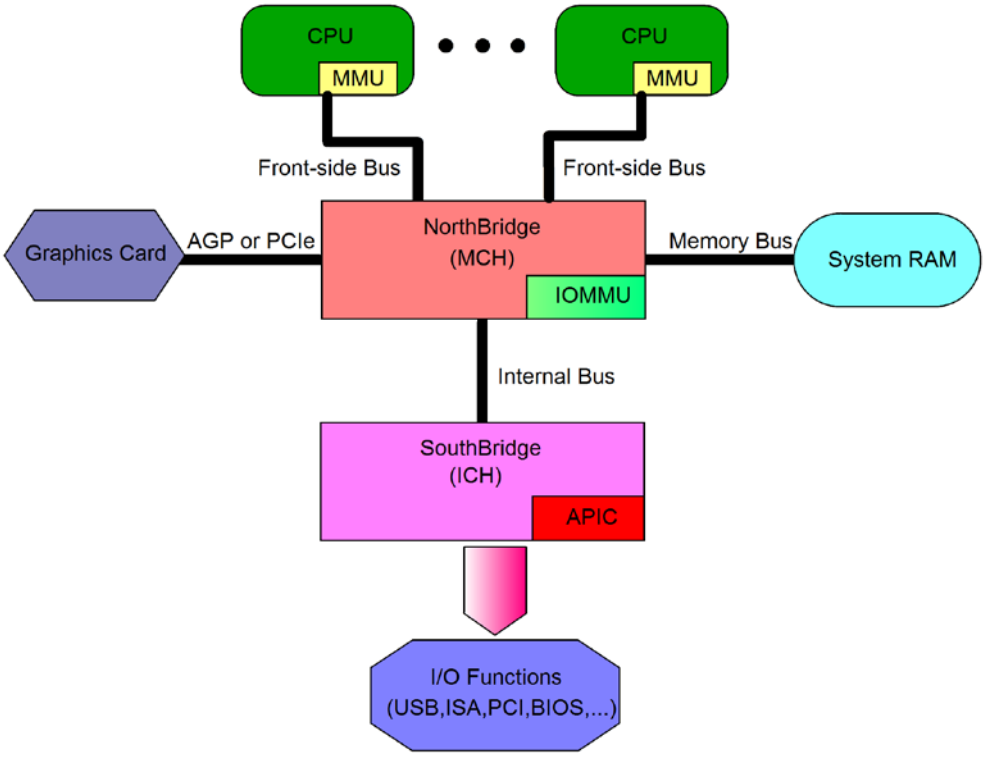


Figure 7. An overview of x86 hardware platform

A northbridge handles communications between the CPUs, system RAM, the graphics card and southbridge. In some modern processors, such as the AMD Fusion and Intel Sandy Bridge, the northbridge functions are integrated into the CPU chip.

Input/Output Memory Management Unit (IOMMU)

An IOMMU translates I/O virtual memory addresses to corresponding physical memory addresses, thereby making DMA by devices safe and efficient [31]. The idea behind an IOMMU is similar to that of an MMU; an MMU enables a CPU to use virtual CPU addresses, while an IOMMU enables devices to use virtual bus addresses.

3) SouthBridge (aka Input/Output Controller Hub (ICH))

A southbridge is a chip that implements a computer's I/O functions (e.g. hard disks, serial port). It also manages access to the non-volatile Basic Input/Output System (BIOS) memory used to store system configuration data [32].

Interrupt Controller (IC)

An IC is a device in a southbridge that routes interrupt signals received from connected devices to a CPU, with the order based on the interrupt priority levels pre-assigned to the input interrupt signals. Many modern machines adopt an Advanced Programmable Interrupt Controller (APIC), which permits more complex priority levels and advanced interrupt request management.

2.4 x86 Virtualization

Popek and Goldberg classify instructions into 3 categories [33]:

1) Privileged instructions

Those that trap if the processor is in user mode and do not trap if it is in system mode, such as instructions which are intended to change the value of a control register.

2) Control sensitive instructions

Those that attempt to change the configuration of resources in the system, such as CLI, which is intended to clear the interrupt flag (IF).

3) Behavior sensitive instructions

Those whose behavior or result depends on the configuration of resources, such as the instruction, INT N, which calls the Nth interrupt handler in the interrupt descriptor table (IDT).

The requirement for a system to be virtualizable is that sensitive instructions should be a subset of privileged instructions. However, 17 sensitive instructions of x86 do not satisfy this requirement; SIDT for example, which is used to store the content of the interrupt descriptor table register. Several types of virtualization have been proposed to solve the problem. The following are the three most popular:

1) Full virtualization

Full virtualization enables an unmodified OS to run in a virtual machine that simulates

the entire hardware environment for a running OS. The basic idea is to trap privileged instructions that are executed in the unprivileged mode in a guest OS, and emulate the behaviours of these privileged instructions in the VMM or the hosted OS. An example of full virtualization is the VMware binary rewriting approach. This approach scans the instruction stream, and marks all privileged instructions that are rewritten to port to their emulated versions. A disadvantage of full virtualization is poor performance caused by the trap-and-emulate overhead.

2) Paravirtualization

Paravirtualization requires a guest OS to be modified to run in a virtual machine. The idea is to modify the guest OS to run in ring 1, so that the VMM can run exclusively in ring 0. The guest OS is aware that it is in a paravirtualized environment. A guest OS cannot execute any privileged instructions, such as updating page tables, because it runs in lower priority and its privileged operations are ignored by the VMM. The VMM typically offers a hypercall mechanism for a guest OS to request privileged operations. A disadvantage of paravirtualization is that it is not feasible to virtualize a legacy OS, such as Windows. However, it is still a popular virtualization solution, due to its high efficiency for performing I/O [34].

3) Hardware-assisted virtualization

Hardware-assisted virtualization (HVM) is a special kind of full virtualization. HVM requires support of the underlying CPU for virtualization. Both Intel and AMD have introduced HVM support in processors released after 2007. The fundamental idea of HVM is to add an extra executing mode (which can be thought of as ring -1), such as VMX for Intel processors, which has higher priority than ring 0. An unmodified guest OS continues to run in ring 0, while the VMM stays in ring -1. The VMM traps and emulates privileged instructions for guest OSes. Compared to paravirtualization, HVM performs better with CPU intensive workloads, and not as effectively with I/O intensive workloads [35].

2.5 Types of VMM

Goldberg [36] classifies VMMs into two types:

Type 1- Bare metal VMM (aka native VMM)

Bare metal VMM refers to the type of VMM that runs directly on the underlying hardware. As shown in Figure 8, a guest OS runs on a higher level (lower priority) above the VMM. The management of Type 1 VMM, such as the creation of a virtual machine or domain, is implemented by a management console, which is an additional piece of software. The management console usually operates in command mode. Xen is a classic example of type 1 VMM. The Xen VMM is used in our project, and will be discussed in Section 2.6.

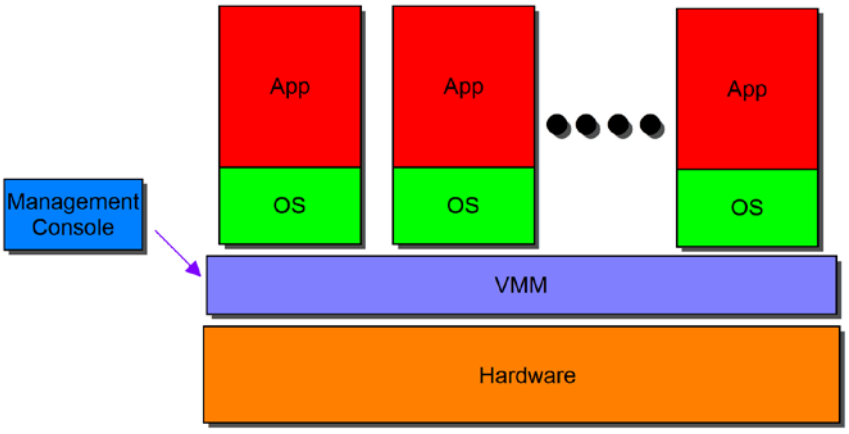


Figure 8. A bare metal VMM

Type 2- Hosted VMM

The type 2 VMM is a hosted VMM, because it needs to reside in an existing OS environment (known as a hosted OS). As shown in Figure 9, a hosted OS is directly installed on the underlying hardware. A type 2 VMM is installed in the hosted OS as an application, and instances of OSes are installed on the type 2 VMM. The hosted OS typically provides a convenient procedure for users to manage a type 2 VMM via a graphical user interface (GUI). A type 2 VMM usually has lower performance than a type

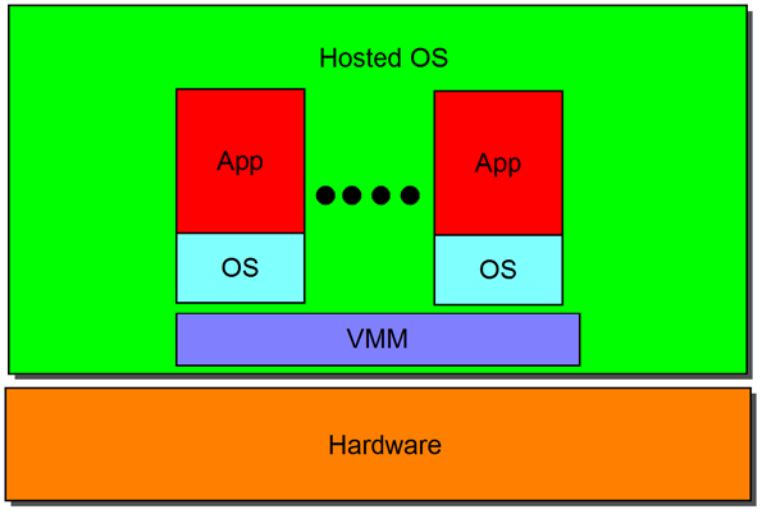


Figure 9. A hosted VMM

1 VMM. The Kernel-based Virtual Machine (KVM) is an example of a type 2 VMM. Under KVM, virtual machines are created by opening a device node (/dev/kvm) [37]. Creating and running virtual machines is achieved through ioctl() system calls. Currently, KVM only supports full virtualization. All I/O accesses are forwarded to the user space of the hosted OS where the Quick EMUlator (QEMU) [38] is used to emulate their behavior.

2.6 Introduction to the Xen VMM

Xen, a paravirtualizing open-source VMM, was first released as a paravirtualization solution in 2003, mainly for x86 platforms [39]. Xen has supported HVM since Intel and AMD processors began supporting virtualization. As mentioned in Section 2.3, HVM allows unmodified OSes to run over a VMM, but it results in low I/O performance. Our project adopts the Xen VMM to build a paravirtualized environment. An overview of the background knowledge relating to the Xen VMM is presented in this Section.

2.6.1 Priority Levels in the Xen VMM

As introduced in Section 2.3, on x86 platforms ring 0 is set as the highest priority, and ring 3 as the lowest priority. Ring 0 is designed for an OS to stay, and applications usually run in ring 3. In a paravirtualized environment each running OS needs to be modified to run in ring 1, while the Xen VMM runs exclusively in Ring 0, guarding accesses to all privileged operations and hardware resources. In this case, an OS is not permitted to implement privileged operations they could implement before, such as updating page tables. Instead, the Xen VMM offers a hypercall mechanism, which is the only method for these running OSes to interact with the Xen VMM to request privileged operations.

2.6.2 Domains

Xen refers to each running virtual machine as a domain. Xen supports one privileged domain, domain 0 (Dom0), and multiple unprivileged domains (DomUs). As the Xen VMM has the highest priority, and is responsible for all the privileged operations in the entire paravirtualized system, the Xen VMM needs to be as bug-free as possible. Therefore, the Xen VMM does not include any device drivers. Dom0 is usually delegated as the driver domain, which includes all the drivers for the underlying hardware. This is why Dom0 is a special domain, with higher privilege than other domains.

The unprivileged domains are not given direct access to the underlying hardware; instead they require the use a split device driver model. As shown in Figure 10, Dom0 has backend drivers installed, and each DomU has frontend drivers. In order to access the underlying hardware, a virtual frontend driver in a DomU communicates with the related virtual backend driver in Dom0 (driver domain), which is usually achieved by using shared memory. The latter then forwards the received I/O requests to the corresponding real device driver. Dom0 in a split device driver model typically consists of a backend

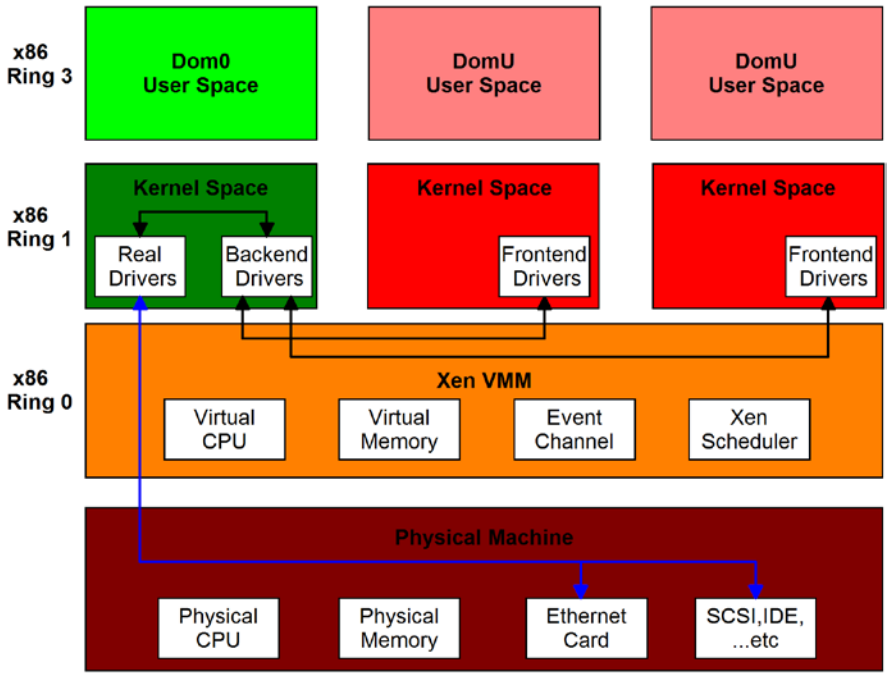


Figure 10. An overview of Xen-based paravirtualized environment

driver, a device driver and a multiplexer that handles multiplexing multiple requests from DomUs to access the shared hardware.

The Xen VMM recently added direct I/O access support for a domain, also known as device passthrough (e.g. PCI passthrough). The passthrough method can give a particular domain near-native I/O performance, but it violates the concept of sharing in virtualization. It also causes security problems on systems without an input/output memory management unit (IOMMU) [40].

2.6.3 Memory Management

The memory used in modern operating systems has already been virtualized, and each process has its own address space. From a process prospective, it assumes that it is the only process running on the machine, and that it has access to the entire memory space. The Xen VMM provides a pseudo-physical memory model to realize the isolation between

different domains.

Figure 11 depicts a pseudo-physical memory model of the Xen VMM. The four colored blocks on top represent four virtual addresses used in a domain. The virtual addresses are first translated into pseudo-physical addresses, and the pseudo-physical addresses are then translated into real physical addresses. A guest OS allocates and maintains its own page tables, but the page tables are marked with read-only. Updating the page tables requires the OS to use an explicit hypercall. The Xen VMM validates all the updates that it deems safe [41]; for example, an update request to map a machine page belonging to another domain will fail. The Xen VMM maintains a globally readable machine-to-physical table, which records the mapping from machine (real physical) page frames to pseudo-physical ones [42]. Conversely, a guest OS is supplied with a read-only (pseudo)physical-to-machine table, which is mapped into its address space through the shared info page. The physical-to-machine table implements the translation of a pseudo-physical address to a machine address. This table is referenced when a guest OS uses a hypercall to update its page table.

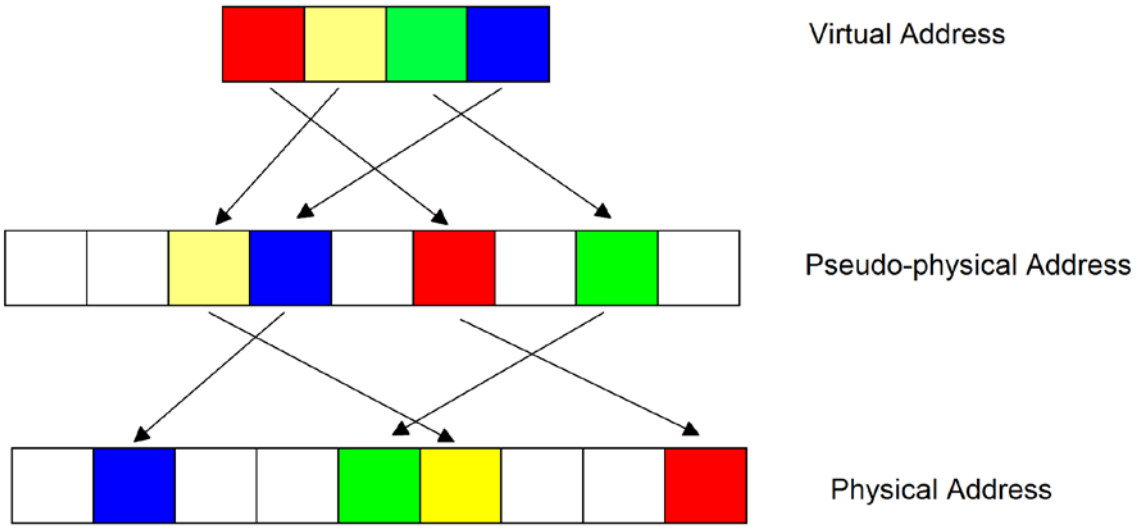


Figure 11. A pseudo-physical memory model

2.6.4 Mechanisms

The Xen VMM provides numerous mechanisms that allow domains to run correctly in the virtualized environment; the aforementioned split device driver model is an example of such a mechanism. The following are some other commonly used mechanisms supplied by the Xen VMM:

1) Event Channel Mechanism

An event can be understood conceptually as a virtual interrupt. An interrupt is an asynchronous notification delivered to the machine hardware, while an event is an asynchronous notification delivered to a virtual machine (domain) by the Xen VMM or another domain. A domain needs to request an event channel from the Xen VMM to deliver events to another domain. After obtaining an event channel, both of the two domains need to bind the event channel to an interrupt number, and register an interrupt handler with that interrupt number so that the corresponding interrupt handler will be invoked when an event is received. An example use of events is a split device model in which the frontend and backend drivers notify each other when an I/O request or response has been sent.

2) Grant Table Mechanism

The grant table mechanism allows a domain to share or transfer its memory pages, with or to another domain. Each domain has its own grant table, which is shared with the Xen VMM. The table is a data structure storing the information used for memory sharing between domains, such as which operation a grantee domain is allowed to perform on the granter's offered memory. Entry of such a data structure in a grant table is identified by a grant reference which is an integer index. When one domain wants to share its memory with another domain, a grant reference corresponding to the data structure describing that shared memory must be passed to the grantee by the granter via some out of band mechanism, such as XenStore.

3) *XenStore*

The XenStore is a hierarchical storage system maintained by Dom0, and shared among all DomUs. It is mainly used as an extended method of transmitting small amounts of information between domains [34]. For example, the aforementioned grant reference can be put in the XenStore by one domain, and obtained by another domain.

2.6.5 Xen Scheduler

The default and most widely used scheduler in the Xen VMM is the credit scheduler, which focuses on allocating CPU resources in a rational manner to each domain according to their pre-assigned weight. The scheduler adopts vCPUs (virtual CPU) as scheduling entities, and each domain can be assigned one or more vCPUs. The default scheduling time quantum is 30ms; that is, scheduling decisions are made every 30ms. The scheduler debits the credits of each running domain on a tick period (10ms) basis. Domains that have consumed all of their allocated credits will be put into the OVER FIFO (first-in-first-out) queue at the following scheduling point, which means they are over scheduled. While they can remain in the UNDER FIFO queue but be moved to the end of the queue if they still have credits remaining. Once the sum of all the active domains becomes negative, the scheduler will allocate new credits to all the domains at the next scheduling point according to their weight. Domains in the OVER queue will not be selected to run unless there are no domains in the UNDER queue ready to run [43]. Therefore, some domains can use more than their share of processor resources but only on the condition that the processor would otherwise have been idle.

However, domains that only have I/O-bound tasks are usually blocked when they are waiting for I/O responses. When an event is sent to a blocked domain, the scheduler will activate the domain and put it into the UNDER queue. Hence, the de facto I/O response latency is largely determined by the activated domain’s position after it is inserted in the queue. Accordingly, a BOOST state, which has higher priority than OVER and UNDER states, is introduced by the credit scheduler to reduce the latency caused by scheduling

delay. With the BOOST mechanism, each time a blocked domain is activated for a pending event it will be assigned a BOOST priority, and allowed to preempt the running domain. For impartiality, the priority of an activated domain will not be boosted if it is in the OVER queue, which implies that it had both I/O and CPU bound tasks, and the previously allocated sharing resources have been totally consumed. Consequently, domains running only I/O bound tasks can realize lower I/O responsiveness, due to the BOOST mechanism [43].

2.7 User Space and Kernel Space

System virtual memory in a Linux OS can be split into two distinct regions: user space and kernel space. User space refers to the address space (from 0 to 3GB in a 32-bit Linux) where user processes run. An OS usually runs in kernel space (from 3GB to 4GB in a 32-bit Linux). An OS kernel runs in ring 0, and has higher priority than user processes which runs in ring 3. Codes running in kernel space, such as device drivers, are allowed to access the underlying hardware.

A user space process usually requests services from an OS kernel via system calls. There are more than 300 system calls in the Linux kernel. System calls are realized by the software interrupt, INT 0x80, which causes the CPU to switch to kernel mode. Each system call is identified by a unique system call number stored in the system call table. The system call number is passed to the EAX register, and the parameters relating to the system call are pushed into the stack before the software interrupt is called, and the related handler is then invoked.

In Xen-based paravirtualization, system calls cannot be executed directly by the guest OS since it runs in ring 1, which does not have high enough priority to execute system calls. In this case, the interrupt handler is installed in the Xen VMM that runs in ring 0. As shown in Figure 12, when interrupt 80h is raised, the execution jumps to the Xen VMM, which then redirects control back to the guest OS [34]. This extra layer of direction allows

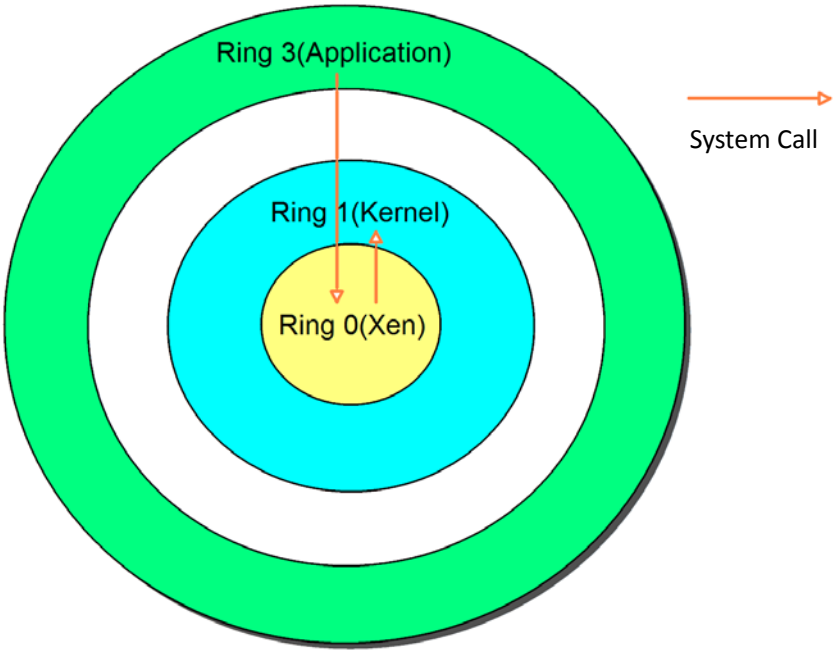


Figure 12. System calls in a paravirtualized system

unmodified applications to run at the expense of a small speed penalty.

Memory mapping is the only way to transfer data between user and kernel space that does not involve explicit copying, and it is the fastest way to handle large amounts of data transfer between user and kernel space [44]. Memory mapping enables a piece of physical memory shared by the user space and kernel space. Data transferred with memory mapping mechanism usually has no 'control' messages with it. This can cause problems; for example, how would the kernel know that a user application has finished its use of the shared memory. Therefore, some synchronization mechanism is required when memory mapping is adopted for user-kernel space data exchange. The following are two classic mechanisms to achieve synchronization:

1) Sleeping

A user process can be put to sleep, which labels it as being in Sleep state and removes it from the run queue. Until the process is awakened in some way, it will not be scheduled to run on a CPU. Sleeping typically gives a rise to a process context

switch. According to [45], certain circumstances must be considered in order to use sleeping in a safe manner:

a) a driver should not sleep while holding a spinlock, seqlock, or read-copy-update (RCU) lock;
b) a process should not be allowed to assume the state of the system after awakened; that is, it must recheck the condition for accessing the shared resource; and,
c) there should be at least one event that will awaken a sleeping process, otherwise the process should not be put into Sleep state.

2) Spin locks

The most common lock in the Linux kernel for thread synchronization is the spin lock [46]. A spin lock can only be held by one thread, and other threads not holding a spin lock will be prevented from accessing the shared resource. A thread blocked by the spin lock will spin repeatedly, executing a tight instruction loop until the lock is released by the thread holding it. Spin locks avoid the overhead of context switches that is caused by sleeping. Using spin locks for synchronization becomes inefficient when the waiting time is extended, because the busy waiting spin locks waste CPU cycles (sleeping does not have this issue).

2.8 An Overview of PCI Express

2.8.1 PCI-based Systems

Peripheral Component Interconnect Express (PCIe) devices can be used in PCI architecture based systems. The PCI architecture is designed to replace the Industry Standard Architecture (ISA) due to its better performance when transferring data between a computer and its peripherals, and simpler adding and removing peripherals [45].

Most of today's x86 machines are PCI based system (see Figure 13 example). The host bridge, which is actually the northbridge chipset in Figure 6, connects the processor to the first PCI bus (PCI bus 0). The PCI-to-ISA bridge, which is actually the southbridge chipset in Figure 7, handles all the computer's I/O functions. The two PCI buses are connected by a PCI-to-PCI bridge. These bridges are also considered as devices connected to the PCI buses.

One PCI system can support up to 256 PCI buses, and each bus can host a maximum of 32 devices, so a PCI system is essentially as a tree structure. Every device in the tree is identified by a bus number, a device number and a function number. When the system boots, the firmware (BIOS) will walk through the PCI tree and probe all the PCI devices in the tree. Once the BIOS detects a device, it will configure the device, and allocate an appropriate address space (of the processor) for the memory in the PCI device that needs to be mapped. Thus, when an OS kernel gets the control of the machine, the PCI devices have already been initialized. On Linux, a user can get all the PCI devices' information in the /sys/bus/pci/devices directory.

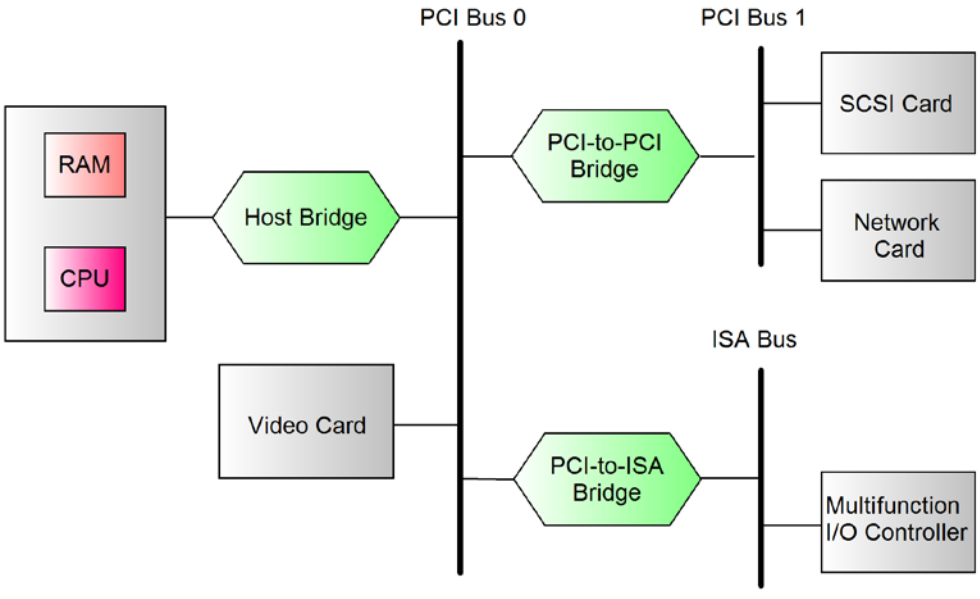


Figure 13. An example of PCI-based system

2.8.2 PCI Express

PCIe is a high speed serial bus standard that supports full-duplex communication links between any two components. PCIe supports three address spaces: configuration address space, I/O address space and memory address space. The configuration address space, which provides configuration information such as device ID and status, is usually accessed by the BIOS when a system boots. The I/O address space and memory address space are used by a device driver, and I/O address space is currently deprecated to be used. A PCIe device's memory can be mapped to a host machine address space through a base address register (BAR), and a device can have up to 6 BARs (BAR 0 to 5). The BIOS or OS determines what address to assign to the device, and the address is stored in a BAR. The device then uses this address to perform address decoding. From the perspective of the CPU, accessing the memory mapped BAR is just like accessing the regular system RAM.

PCIe uses a packet-based protocol for information exchange between the transaction layers of the two components that are communicating over the link [47]. The header of a PCIe packet can be configured to be 3-double-word (3-DW) long for a 32-bit addressing mode, or 4-DW long for a 64-bit addressing mode. The header includes information such as the length of data payload, format of a packet and type of a packet. Though the PCIe 2.1 standard illustrates that the length of data payload can be from 1-DW to 1024-DW, most current motherboards support 4-DW. When a system boots, the two components will communicate using 4-DW payload even though the PCIe device is capable of communicating with a longer payload.

PCIe supports two types of interrupts: legacy interrupt and message signaled interrupt (MSI). Legacy interrupt is used for compatibility purposes by emulating a PCI INTx to signal an interrupt to the system interrupt controller. The MSI mechanism delivers interrupts by performing memory write transactions, and it allows a device to allocate up to 32 interrupts. Each interrupt is assigned dedicated 16-bit data, and the dedicated data are sent to a dedicated memory address to trigger interrupts.

2.9 Direct Memory Access

Direct memory access (DMA) is a technique of transferring data between a peripheral device and the host system memory, independent of the CPU. With the traditional programmed input/output (PIO) method, the CPU is busy moving data from one point to another. The operating frequency of the memory can also be a data transfer bottleneck. Since the CPU is not in charge of data transferring using the DMA technique, it is free to execute other tasks during the data transfer period. There are generally two types of DMA implementations, system DMA implementation and bus master DMA implementation.

System DMA implementation is mainly used in the ISA bus. The implementation requires that the bus have a central DMA controller, which is shared by all the devices on the bus. System DMA implementation is not commonly used today, and very few root complexes and operating systems support it [47].

Bus master DMA implementation is widely used in PCIe based systems. This implementation requires that each PCIe device have its own on-chip DMA controller. The devices can gain from becoming masters on the PCI bus. Thus, the peripheral device reduces the load on the on-host DMA and at the same time operates more efficiently [48].

For convenience in the thesis, ‘DMA read operation’ means that the DMA controller reads data from the host system’s memory, and ‘DMA write operation’ means that the DMA controller writes data to the host system’s memory.

Chapter 3

Basic Framework Design

The system design can be divided into two major sections: the FPGA accelerator design and the FPGA virtualization on an x86 server. The accelerator design is a conventional FPGA hardware design work, which is normally implemented by using HDL and/or Intellectual Property (IP) cores. The virtualization work is software-related development that involves the design of a frontend and backend driver and a real device driver for the FPGA accelerator, as well as the proposed coprovisor design.

The overall layout of the pvFPGA system design is shown in Figure 14. A DomU needs to pass a request to Dom0 to access the FPGA accelerator. The Dom0 is responsible for multiplexing requests from multiple DomUs for access to the FPGA accelerator.

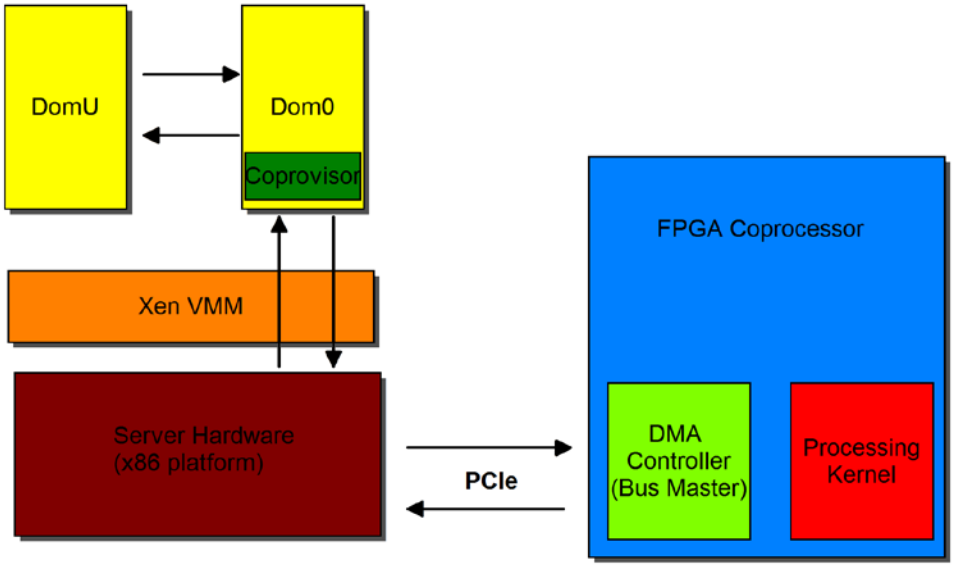


Figure 14. The layout of pvFPGA design

3.1 FPGA Hardware Accelerator Design

3.1.1 Design with One DMA Channel

We adopted the PCIe interface [49] as the communication channel, and used the DMA technique for efficient transfer of data to and from the host server memory. Unlike traditional ISA devices, there is no central DMA controller for PCIe components. To address this, we designed a bus master DMA controller on the FPGA accelerator.

With only one DMA channel [47], we need to utilize additional memory (e.g. DDRII memory) to store data when the computing procedure involves large amounts of data. We used the DDRII memory controller IP core [50] from the Xilinx Corporation. The overall FPGA accelerator design is shown in Figure 15. The on-chip FIFO buffer is a memory queue IP core supplied by Xilinx for applications requiring in-order storage and retrieval [51]. The input and output of a FIFO buffer can operate at different frequencies (in the figure, same color modules operate at the same frequency). Our design supports two user applications running on the FPGA accelerator. An application selector module is responsible for selecting the application to run during runtime. The application selector can be runtime configured by the device driver, which is realized by mapping one of the selector's registers to the kernel space through PCIe BAR 0. The FPGA accelerator works as follows:

1) the application selector is correctly configured by the device driver before the DMA controller starts data transfer;
2) when the DMA transfer starts, all the data from the server system memory are streamed to the FPGA accelerator, and stored in an on-chip FIFO buffer;
3) once the number of data stored in the on-chip FIFO buffer reaches the threshold (determined by the selected application), the selected application starts fetching data from the FIFO buffer to do a computation;
4) the results of the computation are stored in the DDRII memory;

5) when the DMA read from server memory operation ends, a DMA write to system memory operation is initiated;
6) results stored in the DDRII memory are first moved to an on-chip FIFO buffer first, then they are transferred to the server system memory (the use of the on-chip FIFO buffer enables the DMA controller and the memory controller to work at different frequencies).

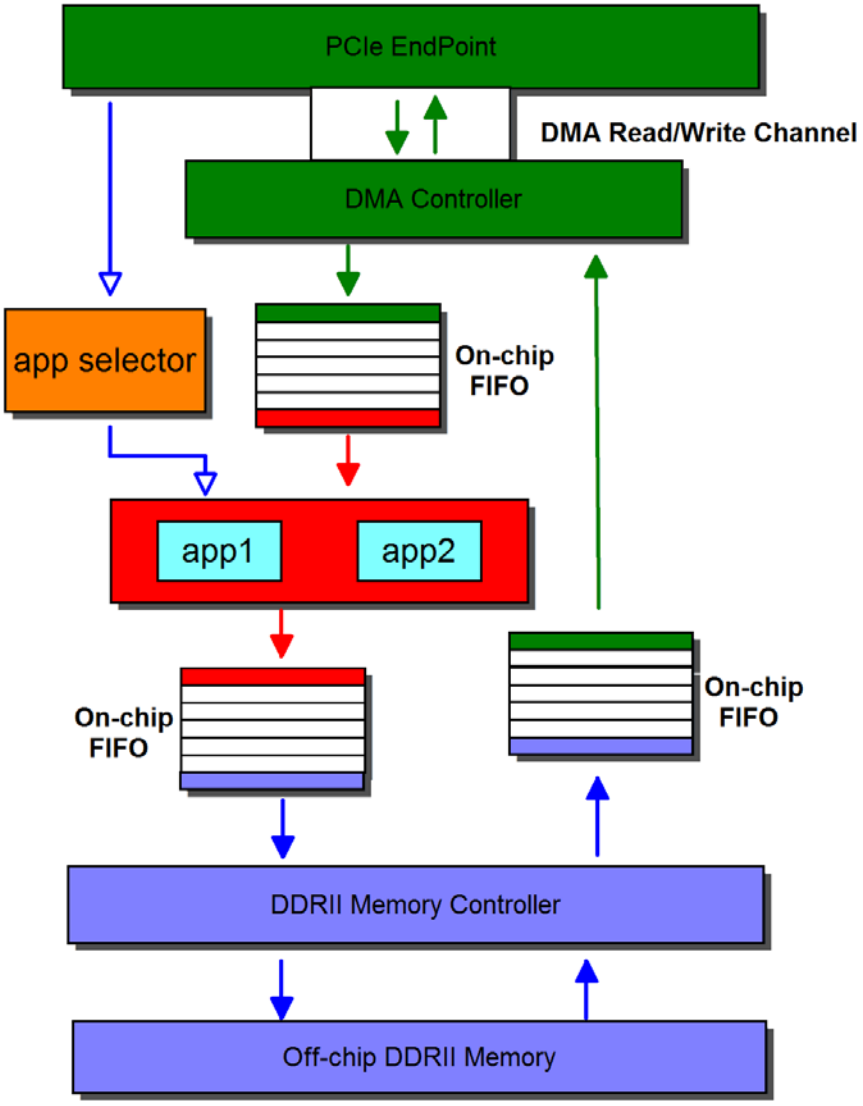
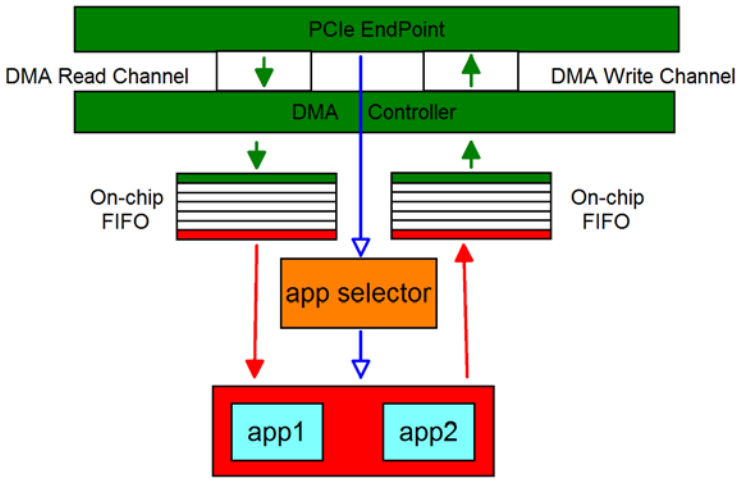


Figure 15. Design of an FPGA accelerator with one DMA channel

3.1.2 Design with Two DMA Channels

We have further upgraded our design by using multiple DMA channels. This allows the off-chip DDRII memory to be removed, since we can do DMA reads from the server memory and DMA writes to the server memory simultaneously. In other words, we take advantage of a streaming pipeline.

Figure 16(a) illustrates a simplified design using 2 DMA channels [52]. Once the data count of the FIFO buffer equals one block of data (4KB in the following experiments), the app starts to fetch data from the FIFO. Figure 16(b) shows the pipelined operations. The overall procedure is conducted in three operations, DMA read, computation, and DMA write. The latency for finishing each of the three operations is identical in Figure 15, but in practice, this is not always the case, since the latency of computation is determined by the application, and the latency of DMA read or write operations is determined by the communication channel.



a) Accelerator Design

b) Pipelined Operations

Figure 16. Design of an FPGA accelerator with two DMA channels

3.2 Design using the Pipeline Technique

With the pipeline technique, some execution latency can be hidden. For example, in the third cycle in Figure 16(b), the DMA read and DMA write operations are implemented simultaneously as the FPGA accelerator is doing a computation, so the latency of the DMA read and write is hidden.

The pipeline shown in Figure 16(b) is similar to a classic RISC (Reduced Instruction Set Computer) pipeline, where all the operations have identical execution latency. In the following discussion, we assume the computation latency is $\mathbf{T_C}$, and the DMA data transfer latency of one block of data is $\mathbf{T_D}$ (that is, the DMA read latency = DMA write latency= $\mathbf{T_D}$), and the total number of blocks of data is **N**, then we get:

Without the pipeline technique, the overall turnaround time (overall system's implementation latency) is

$$(2\mathbf{T_D} + \mathbf{T_C}) * \mathbf{N} \qquad \textbf{(F-1)}$$

With the pipeline technique shown in Figure 16(b), the overall turnaround time is

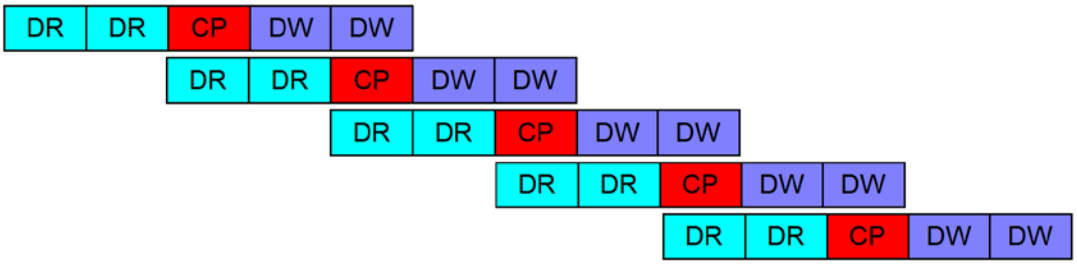
$$\mathbf{T_D} + \mathbf{T_C} + \mathbf{N}*\mathbf{T_D} \qquad \textbf{(F-2)}$$

In order to show that using the pipeline technique produces lower turnaround time, the following equation must be true:

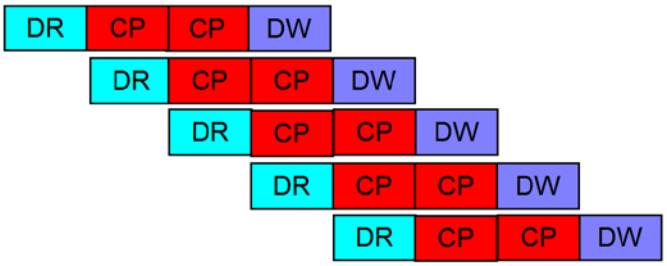
$$(2\mathbf{T_D} + \mathbf{T_C}) * \mathbf{N} > \mathbf{T_D} + \mathbf{T_C} + \mathbf{N}*\mathbf{T_D} \qquad \textbf{(F-3)}$$

Then we get **N>1**, which is the condition that needs to be satisfied for the turnaround time to be reduced by using the pipeline technique. In most designs that use the pipeline technique, all the data can be split into blocks to be transferred (that is, N>1). Therefore, the turnaround time can be significantly reduced using the pipeline technique, as long as more than one block of data needs to be transferred (otherwise, there is no pipelined operations exit).

However, this type of pipeline shown in Figure 16(b) seldom occurs with the design of an FPGA accelerator, because it is uncertain that the computation latency, $\mathbf{T_C}$, is equal to the DMA data transfer latency, $\mathbf{T_D}$. Following are two possible scenarios:



a) An example of Scenario 1 pipelined operations



b) An example of Scenario 2 pipelined operations

Figure 17. Examples of non-RISC pipelined operations

Scenario 1: In Figure 17(a), the DMA data transfer latency is larger than the FPGA accelerator computation latency, that is, $\mathbf{T_D} > \mathbf{T_C}$. One block in the figure occupies one timing unit. DR represents DMA read, and two concatenate DR blocks mean that the DMA read operation occupies two timing units; CP represents Computation on the FPGA; DW represents DMA write.

Scenario 2: In Figure 17(b), the DMA data transfer latency is less than the FPGA accelerator computation latency, that is $\mathbf{T_D} < \mathbf{T_C}$.

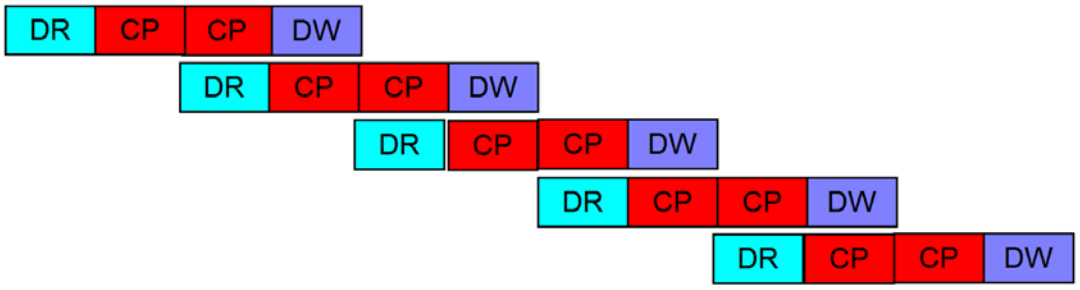
The Scenario 1 pipelined operation can work just like the RISC pipeline technique, and the overall turnaround time can also be calculated by formula (F-2). However, the Scenario 2 pipelined operations shown in Figure 17(b) will cause some problems. The overlapping of two computation operations implies that two blocks of data are contending for one computation module. In other words, when the second block of data arrives in the FPGA accelerator for doing a computation, the first block of data has not completed its computation. This will cause a conflict.

A simple method to solve the conflict problem shown in Figure 17(b) is to increase the operating frequency of the computation module, thereby reducing the computation latency, $\mathbf{T_C}$. But this is not always feasible, because there is always a limit to the maximum frequency of the design. Exceeding the maximum frequency would cause issues such as setup/hold timing violations.

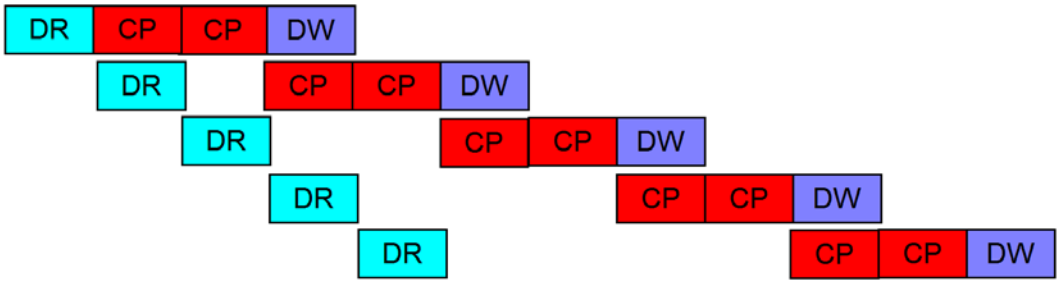
The second method to solve this problem is shown in Figure 18, where the start of next DMA read operation is delayed. This is very difficult to accomplish, because the time to start the DMA read operation needs to be at an accurate time instance when the FPGA is executing a computation. Figure 18(a) shows an example where $\mathbf{T_C}$ equals 2*$\mathbf{T_D}$, and the time to start the DMA read of next block of data can be set when the FPGA finishes the current half block of data computation. However, in practice $\mathbf{T_C}$ could be any factor larger than $\mathbf{T_D}$ (e.g. 1.5 or 1.75). Managing this method is complicated, and varies with different applications because they have different computation latencies.

An alternative to Figure 18(a) is to delay the start of the computation module, as shown in Figure 18(b). It is clear in the figure that the two methods have the same turnaround time. In practice, the technique shown in Figure 18(b) is prone to implement, because the start of next DMA read operation can be just triggered when the first DMA read operation finishes, but the incoming data are not processed immediately when they arrive in the FPGA accelerator. For example, when the second block of data is computed, the third to fifth block of data have already arrived and must be buffered. It can be seen in Figure 18(b) that the gap between DR and CP gets enlarged when **N** increases. Thus, the disadvantage of the method in Figure 18(b) is it requires large memory to buffer the incoming data, and the required size of the buffer grows as **N** increases.

Figure 19 illustrates a third method to address the problem; hide the DMA write operation latency by executing it simultaneously with a DMA read operation. The drawback of this method is that it has longer turnaround time than the other methods. However, the fact that it can be used for general purpose computing is an advantage. That



a) Delay the start of DMA read operations



b) Delay the start of computation operations

Figure 18. Delayed operations in the pipeline technique

is, in practice it can be easily adopted for any situation regardless of the latency difference between the DMA data transfer and the FPGA computation. This type of pipeline satisfies our FPGA accelerator design aim, which is to make the FPGA accelerator capable of accelerating various applications for cloud clients. The overall turnaround time of the pipeline shown in Figure 19 can be calculated by the following formula:

$$T_D + N*(T_C + T_D) \qquad \textbf{(F-4)}$$

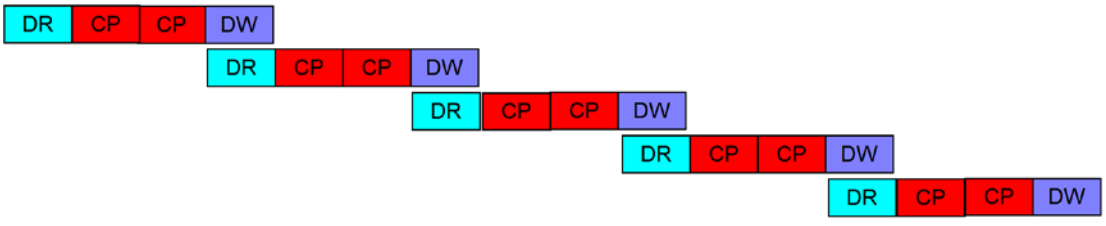


Figure 19. Simultaneous implementation of DMA read and write operations

3.3 Device Driver Design for the FPGA Accelerator

Since communication between the FPGA accelerator and the server is implemented through the bus master DMA controller on the FPGA accelerator, the driver actually controls how the DMA controller functions.

Before a DMA operation begins, the kernel usually needs to allocate a DMA buffer for data transfer purposes. A DMA buffer is a piece of physically continuous memory, and the bus address of the DMA buffer should be passed to the DMA controller. The DMA buffer must be used carefully to avoid cache coherence problems. A common solution is for the CPU not to be allowed to operate on the DMA buffer once it transfers control to the DMA controller for operation on that buffer. When the DMA transfer finishes, the CPU can be notified by an interrupt and regain access to the DMA buffer.

Linux has functions for allocating physically continues memory that can be used as a DMA buffer, such as "__get_free_pages ()". The amount of physically continuous memory available is usually less than 1MB. The DMA controller adopted in our design supports a scatter-gather function, which enables data in one non-continuous block of memory to be moved by means of a series of smaller continuous block transfers. Therefore, the device driver needs to allocate several small physically continuous memory fragments (4KB in our design). These 4KB physically continuous memory fragments make up a DMA buffer.

The information about the operation of transferring one continuous memory fragment, such as the start of a new transfer, is contained in a buffer descriptor. The device driver is responsible for allocating and organizing the buffer descriptors, and handing them over to the DMA controller. An example of using buffer descriptors is shown in Figure 20. The driver is not allowed to access the buffer descriptors once the DMA implementation begins using that descriptor. The bus address of the first buffer descriptor needs to be passed to a specific register in the DMA controller before the DMA transfer begins.

The configuration registers of the DMA controller are mapped to the CPU address

space through PCIe BAR 0. Configuring these mapped registers (e.g. resetting the DMA controller, enabling interrupts, initiating the start of data transfer) would change the status of the DMA controller.

The device driver also needs to have support for implementing the pipelined operations. Take the pipeline type shown in Figure 19 as an example, each time the FPGA accelerator finishes a computation, it sends an interrupt to the server. The device driver is responsible for initiating the DMA controller to send back results (DMA write) and fetch new data from the host memory (DMA read) simultaneously.

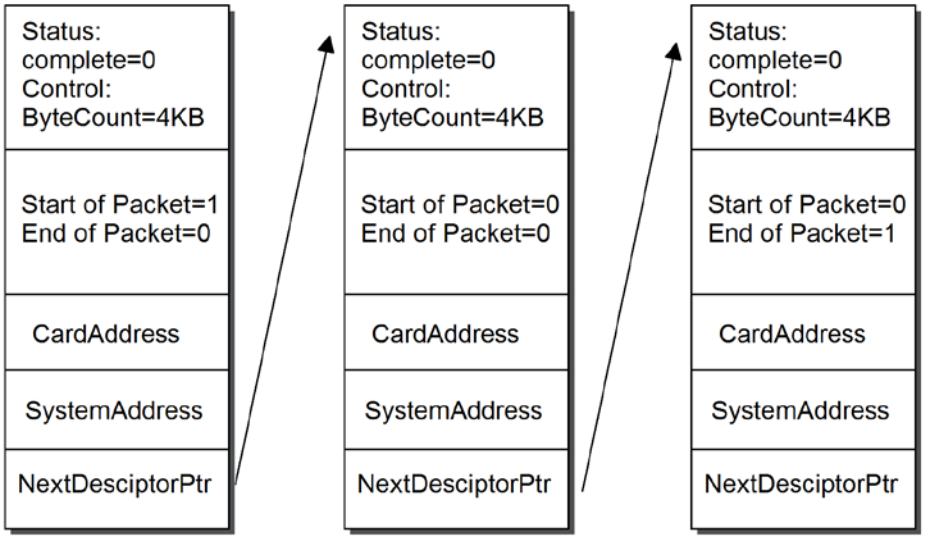


Figure 20. DMA buffer descriptors

3.4 FPGA Virtualization

3.4.1 Data Transfer

Using an FPGA hardware accelerator requires transfers of tens of kilobytes to Gigabytes of data between the server and the FPGA accelerator. The overhead, compared with a native machine (without the VMM layer), is caused by inter-domain

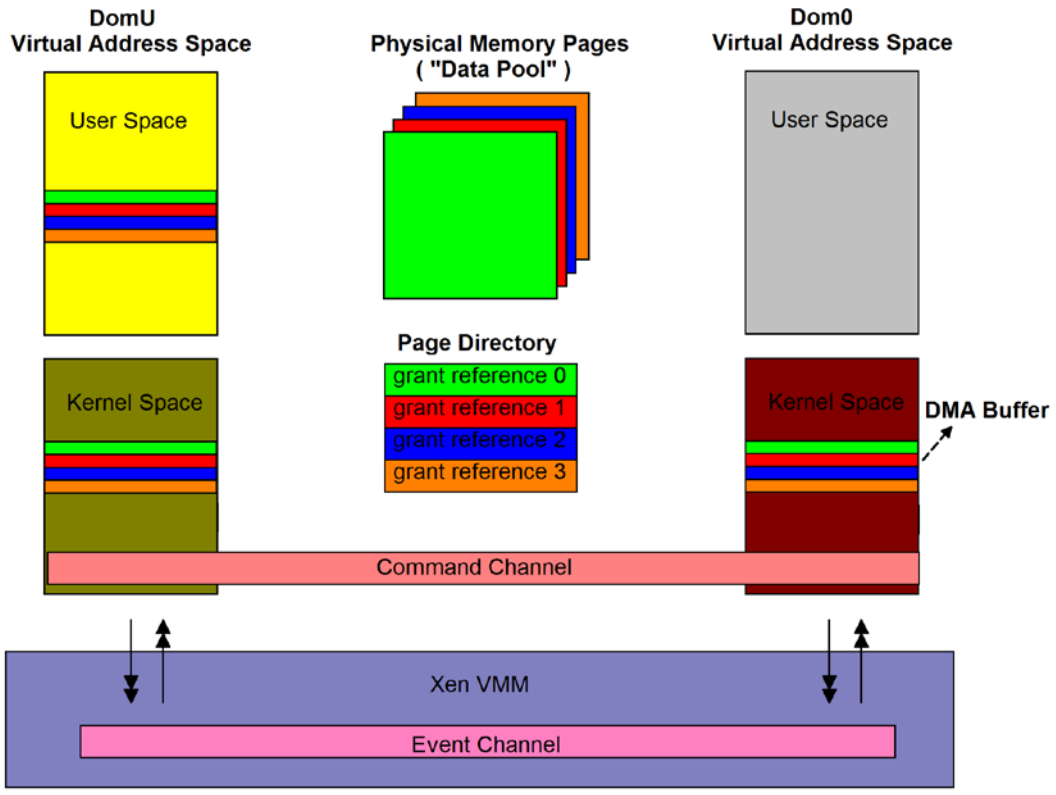


Figure 21. Inter-domain communication design of pvFPGA

communication latency; that is, the time consumed by transferring large amounts of data from a DomU to Dom0 via the split device driver model supported by the Xen VMM. The aim of this part of the design is to access the FPGA accelerator from a DomU with minimal overhead. In addition to inter-domain communication latency, we also need to consider how the data could be efficiently transferred from a user application (residing in user space) to the frontend driver (residing in kernel space) in a DomU.

The challenge of designing efficient inter-domain communication is not unique to FPGA virtualization. Some shared memory based techniques, such as IVC [53], Xway [54], XenLoop [55] and Xensocket [56], have tackled the problem of network communication, but their designs are specific to network packets. The shared memory mechanism has also been used in recent GPU virtualization solutions [11, 14] for inter-domain communication, resulting in low overhead. These solutions are specifically

designed for intercepting and redirecting API calls for a GPU accelerator. Here, we introduce a shared-memory mechanism for pvFPGA which focuses on low overhead data transfer for an FPGA accelerator. The comparison of pvFPGA with recent GPU virtualization solutions is detailed in Chapter 6.

Figure 21 shows the inter-domain communication design of pvFPGA. A DomU kernel allocates a group of memory pages (referred to as a “data pool”) which are reserved for data transfer. A DomU’s user process can map the data pool into its virtual address space so that it can operate directly on the shared memory pages. Meanwhile, the data pool is also shared with Dom0 kernel through the grant table mechanism. To share the data pool, the grant references of these shared memory pages in a data pool are filled in an extra shared memory page, which acts as a page directory. The directory is shared with Dom0 first, and once Dom0 gets the directory page, it is ready to map the provided data pool. Finally, the shared memory pages in the data pool are exposed to the bus master DMA controller residing in the FPGA accelerator. Since the DMA controller referenced in our FPGA accelerator design supports a scatter/gather function, the shared memory pages allocated in the DomU kernel space in the first step are not necessarily physically continuous. Similar to the data pool, the command channel is a memory page shared between a DomU (also mapped by a user space process) and Dom0, and is used for transferring command information. Our design currently uses the command channel to transfer two command elements: 1) the application number that the DomU requires to use on the FPGA accelerator, and 2) the size of the data in the data pool that must be transferred to the FPGA accelerator. Figure 22 presents the dataflow in pvFPGA; the steps are as follows:

1) An application in a DomU user space specifies the application number and data size in the command channel, which is mapped to its address space;
2) the application in a DomU user space adds data directly in the shared data pool, which is mapped to its address space;

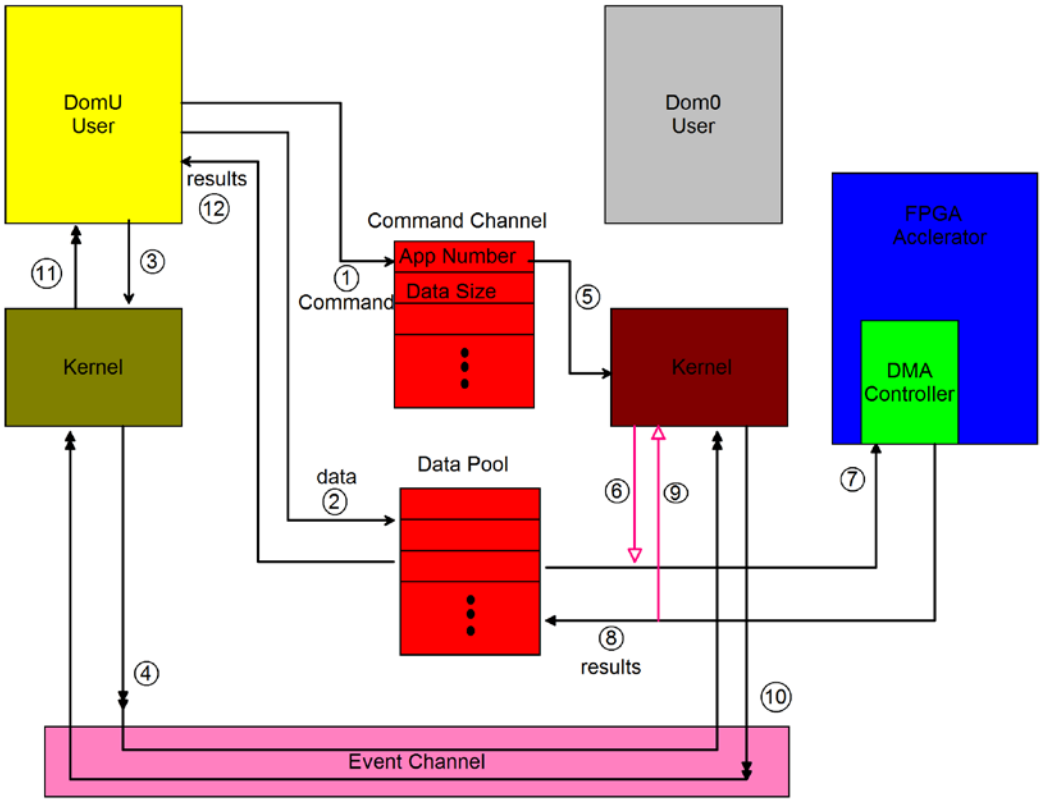


Figure 22. Dataflow in pvFPGA

3) the user application notifies the frontend driver in the DomU kernel space that the data is ready, then goes into the Sleep state;
4) the frontend driver in the DomU kernel space sends an event to the backend driver in the Dom0 kernel space;
5) the frontend driver passes the request to the device driver in the Dom0 kernel space, and the device driver configures the FPGA accelerator through mapped PCIe BAR 0, according to the command information from the command channel;
6) the device driver initiates the start of the DMA controller residing in the FPGA accelerator;
7) the DMA controller transfers all the data to the FPGA accelerator in a pipelined way to do a computation;
8) the DMA controller transfers the results of the computation back to the data pool

(this is part of the pipelined operations in Step 7);

9) the DMA controller sends an interrupt to the device driver when all the results have been transferred to the data pool;
10) the backend driver sends an event to notify the frontend driver that the results are ready;
11) the frontend driver wakes up the user application;
12) the user application fetches the results from the data pool.

All the memory-sharing procedures are finished during the frontend/backend driver loading time. Thus, the only overhead that is caused by the Xen VMM for the inter-domain data transfer is the two event notifications; one notifying Dom0 that data are ready to send to the FPGA accelerator, and the other notifying the DomU that the results are ready to be fetched.

3.4.2 Construction of the Data Transfer Channel

Section 3.4.1 introduced a data pool mechanism that is mapped by a user process in a DomU, and shared by the Dom0 kernel for data transfer. This section discusses how the channel is established.

Since the DMA controller on the FPGA accelerator supports a scatter-gather function, a DomU only needs to allocate a series of scattered memory pages. For example, to allocate a 4MB data pool, a DomU needs to allocate 1024 4KB pages. However, directly allocating 4MB of physically continuous memory will, in most cases, result in a failure due to the OS limit. The data pool is shared with the Dom0 kernel through the grant table mechanism supplied by the Xen VMM. This is achieved through hypercalls, which require the machine frame numbers of these shared pages to be passed as parameters. Since we have 1024 scattered pages, 1024 grant references will be returned by 1024 hypercalls. As mentioned in Section 3.4.1, we use an extra 4KB directory page to hold these grant references. The directory page is shared with the Dom0 kernel first, then the Dom0 kernel

can map the data pool while holding these grant references in the directory page. All this work is done at frontend and backend driver loading time.

In order to enable a process to map the data pool, the allocated pages are represented as one device file in the /dev directory. We rewrite the following system call implementations for user processes to operate on the data pool:

1) mmap()

   In this function, we remap all the allocated 4KB physical pages to the calling process's virtual address space according to the order they are allocated. The start address of the mapped virtual address will be returned. After this call, the user process can directly access the data pool.

2) write()

   This function implements two actions: sending a request to Dom0 and putting the calling process into the Sleep state. This is an interface that a user process notifies the DomU kernel that the data are ready.

The command channel is also mapped by a user process in this way. An alternative method of adding an extra device file for the command channel in the /dev directory is to map it together with the data pool, and reserve the first page for transferring command information. This convention also needs to be known to the backend driver in Dom0.

3.4.3 Coprovisor

The Xen VMM scheduler is only responsible for controlling access to CPUs, while the backend drivers need to provide some means of regulating the number of I/O requests that a given domain can perform [34]. To manage this for pvFPGA, we propose a component that we call a coprovisor, which multiplexes requests from different domains accessing the FPGA coprocessor. For GPU virtualization, the multiplexing work is realized in user space, due to the lack of standard interfaces at the hardware level and driver layer. Precisely, the multiplexer and scheduler are put on the top of CUDA

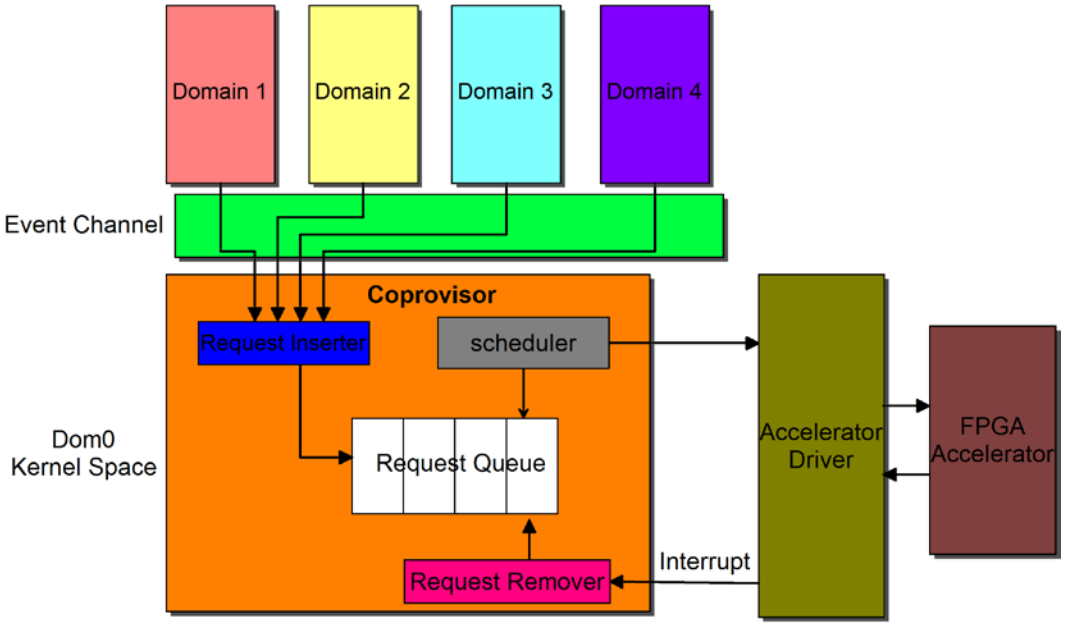


Figure 23. The coprovisor design of pvFPGA

(Compute Unified Device Architecture) runtime or driver APIs [11, 14]. In our case, the coprovisor can perform multiplexing directly at the accelerator driver layer in Dom0.

Some studies [16, 19] use spatial multiplexing to virtualize an FPGA accelerator in a single OS. These works take advantage of the partial runtime reconfiguration capability of an FPGA accelerator. From the perspective of the OS, their modularly designed FPGA appears as more accelerators because more user logic modules on the FPGA accelerator can be swapped in or out during runtime. Their designs only multiplex accesses to the FPGA accelerator for requests from a single OS. The coprovisor in our pvFPGA currently uses time multiplexing, which is the main method used in virtualization to boost hardware utilization. We may incorporate partial reconfiguration to gain more flexibility in pvFPGA in the future. More details about future updates are revealed in Chapter 7.

The architecture of the coprovisor is shown in Figure 23. It consists of four parts: Request Inserter, Scheduler, Request Queue and Request Remover. A DomU notifies Dom0 for accessing the FPGA accelerator via an event channel, so the Request Inserter,

which is responsible for inserting requests from DomUs into the Request Queue, is invoked when an event notification is received at the backend driver. When a request has been serviced, an interrupt from the FPGA accelerator notifies the Request Remover to remove the serviced request from the Request Queue. The Scheduler is responsible for scheduling requests to access the FPGA accelerator through the accelerator device driver. The present scheduling algorithm used by the scheduler is FCFS (first-come,first-served), which ensures that requests from DomUs are extracted in an orderly manner by the scheduler. When a new request is inserted in the Request Queue, or a serviced request is removed from the queue, the Scheduler will be invoked to check if the FPGA accelerator is idle, and if there is a pending request in the Request Queue. If so, the next request of the Request Queue will be scheduled to the FPGA accelerator.

The size of a shared data pool dictates the maximum data transfer bandwidth. For example, a DomU (D1) assigned with a 4MB data pool can transfer a maximum of 4MB data per request to the FPGA accelerator, while a DomU (D2) asigned with a 512KB data pool can transfer a maximum of 512KB data per request. When the two DomUs contend for a shared FPGA accelerator to accelerate applications which need to send more than 512KB data, D2 is slower because it needs to send more requests to complete the entire computations. To provide DomUs with different maximum data transfer bandwidths, the split driver designer can regulate the size of the shared data pool in each DomU's frontend driver, and in the Dom0's backend driver at the frontend/backend driver loading time.

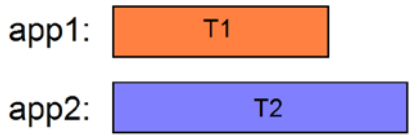
Chapter 4

pvFPGA Amelioration

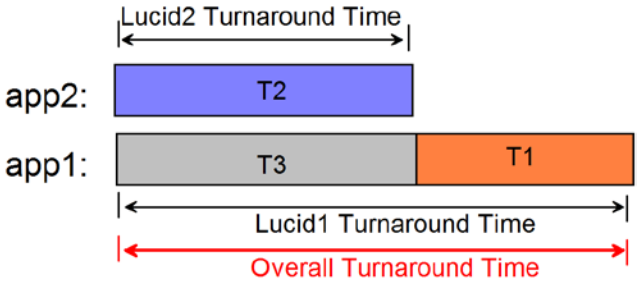
4.1 Analysis

The main framework of pvFPGA has already been discussed. However, it is possible that it might not work efficiently in some cases. Requests from domains are buffered in the Request Queue, and then scheduled to the FPGA accelerator in the FCFS order. We can examine the inefficiency in the following scenario:

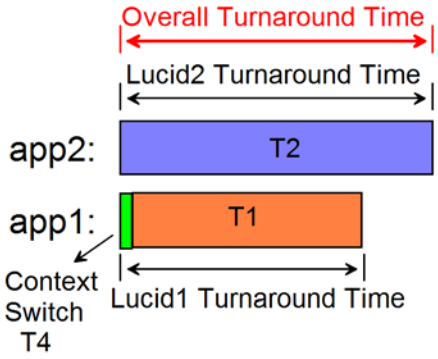
Consider two applications, app1 and app2 running on the FPGA accelerator as shown in Figure 16. The processing time needed by app1 and app2 for processing one block of data is T1 and T2 respectively (see Figure 24(a)). Assume both T1 and T2 are longer than the communication latency of transferring one block of data. Two guest domains, Lucid1 and Lucid2, are requesting access to the FPGA accelerator simultaneously, with Lucid1 requesting app1 acceleration and Lucid2 requesting app2 acceleration. Suppose the Lucid2 request comes first and gets serviced before the Lucid1 request. With our previous design, the turnaround time of Lucid1 was T1+T3 (see Figure 24(b)), where T3 equals T2. The Lucid1 request will not be scheduled to the FPGA accelerator until the Lucid2 request is serviced, which results in a scheduling delay (T3) for Lucid1 to get its request serviced. Even though the DMA read channel (in Figure 16(a)) becomes idle when a block of data is transferred to app2 for processing, the Lucid1 request has to wait to get serviced till app2 on the FPGA accelerator finishes its processing and raises an interrupt to the server. This unnecessary scheduling delay can be eliminated by implementing a DMA context switch. More precisely, once the Lucid2 request finishes its use of DMA read channel, an immediate context switch to the Lucid1 request will be implemented by the coprovisor.



a) Processing time of app1 and app2



b) Overall turnaround time with previous design



c) Overall turnaround time with a DMA context switch

Figure 24. Analysis of request turnaround time

As shown in Figure 24(c), the overall turnaround time will be reduced to T2, and the turnaround time of Lucid1 is T1 plus the context switch overhead T4. The DMA context switch overhead is minor, because only one parameter (the bus address of the next DMA buffer descriptor) needs to be loaded to the DMA controller. Therefore, both Lucid1 and the overall turnaround time will be reduced through a DMA context switch. This will be shown in our experiments in Section 5.6.

4.2 FPGA Accelerator Design Modification

4.2.1 App Controller

Both app1 and app2 in Figure 16(a) could be active on the FPGA accelerator at the same time. The processing latency of app1 and app2 is determined by the specific accelerator applications. So, both app1 and app2 could possibly finish at the same time, and contend for the DMA write channel. The work of multiplexing DMA write channel is implemented in hardware by the FPGA accelerator. Clearly, the app selector in Figure 16(a) which is responsible for selecting apps on the FPGA accelerator is not adequate. Here, we replace it with a more advanced module, an app controller. The app controller performs four functions:

1) Directing the input streaming data from the DMA controller to app1 or app2;
2) Multiplexing app1 and app2 to use the DMA write channel;
3) Maintaining the accelerator status word (will be discussed later); and
4) Raising an MSI interrupt when required.

The first function is the same as that performed by the previously proposed app selector. The driver will direct the app controller to send the input block of data to either app1 or app2, by configuring the accelerator status word (ASW) before initiating the start of a DMA read operation. Also, the app controller needs to multiplex requests from app1 and app2 to use the DMA write channel. Since initiating a DMA write operation is also implemented by the driver, the app controller needs to maintain the ASW status, and ensures it visible to the driver.

4.2.2 Accelerator Status Word

The introduction of the ASW enhances the interaction between the driver and the FPGA accelerator. More precisely, the driver gets to know the status of the FPGA

Table 1. Design of the ASW

31----------4	3	2	1	0
Reserved	AppFinState	AppFin#	DMARState	DMAR#

Table 2. Bits in the ASW

Bit	Functions
Bit 0	0-app1 is selected for the following block of data; 1-app2 is selected for the following block of data. Written by the driver.
Bit 1	This bit is set by the FPGA accelerator when one block of data finishes using the DMA read channel. An interrupt is raised after setting this bit. Read and Cleared by the driver.
Bit 2	0-app1 finishes its processing for one block of data; 1-app2 finishes its processing for one block of data. Read and Cleared by the driver.
Bit 3	This bit is set by the FPGA accelerator either when app1 or app2 finishes a computation. An interrupt is raised after setting this bit. Read and Cleared by the Driver.
Bit 4-31	Reserved for future updates.

accelerator through the ASW, and from the FPGA accelerator prospective, the ASW tells the FPGA accelerator what the driver needs. The design of the ASW is shown in Table 1, and Table 2 describes the functions of each bit in the ASW. With the help of the ASW, interrupts that are raised by the FPGA accelerator as different events occur can be distinguished. When a block of data finishes using the DMA read channel, the app controller will set Bit 1 of the ASW and raise an interrupt; for convenience, we call this a type 1 interrupt. When an app (either app1 or app2) finishes a computation, the app controller will clear (if app1 finishes a computation) or set (if app2 finishes a computation) Bit 2 of the ASW, set Bit 3 of the ASW, and raise an interrupt. We call this a type 2 interrupt in the following description.

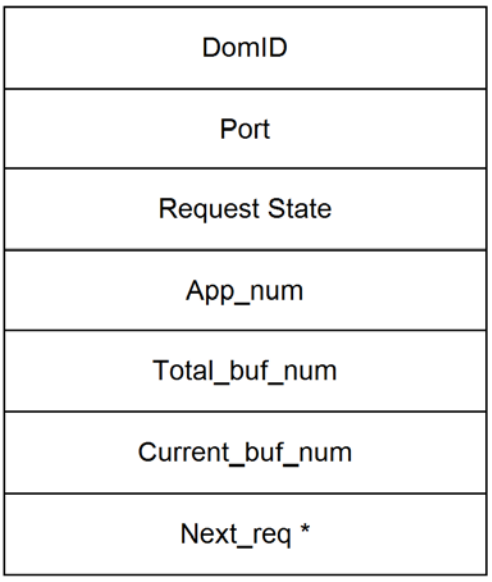


Figure 25. Design of a request control block

Table 3. Description of a request control block

Member	Description
DomID	Denotes the ID of the domain that the request comes from
Port	The port number of the event channel. It is used to notify the request's domain through the corresponding event channel.
Request State	The three possible states of a request: DMAREAD, DMAWRITE and DMAFIN.
App_num	Specifies which app the request needs to use on the FPGA accelerator. 0-app1, 1-app2
Total_buf_num	The total number of buffers used by the request. The size of one DMA buffer is 4KB.
Current_buf_num	Specifies the number of current buffer that needs to be transferred to the FPGA accelerator.
Next_req	A pointer points to the next request.

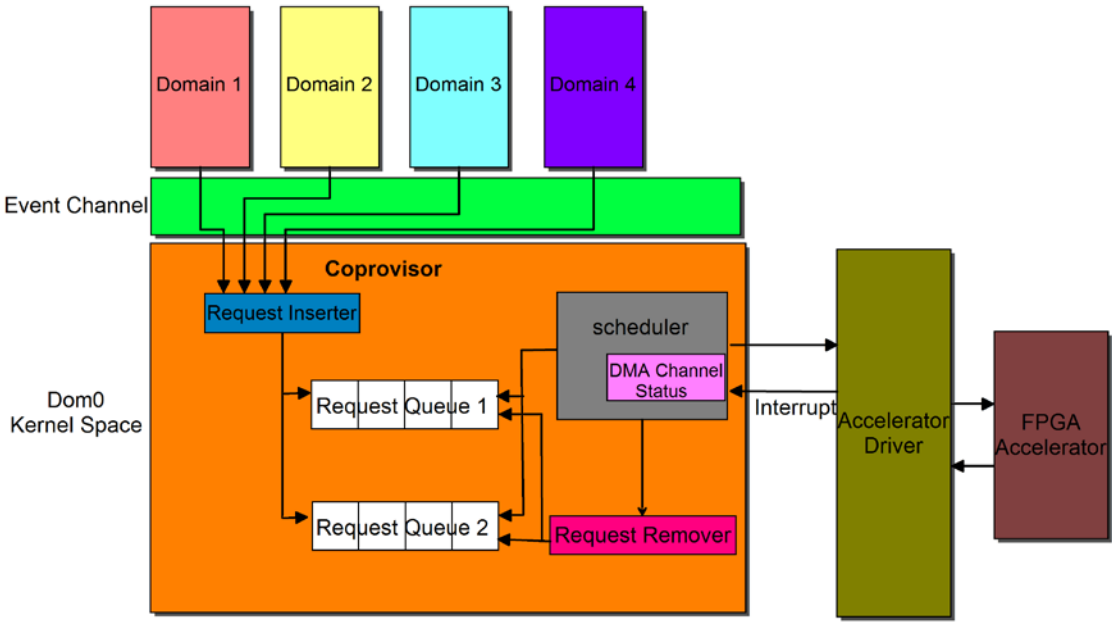


Figure 26. The improved coprovisor design

4.3 Coprovisor Design Modification

In Section 3.4.3, we proposed a coprovisor that is responsible for scheduling requests from DomUs to access the FPGA accelerator. We need to modify the coprovisor design to support DMA context switches. Before discussing the details of modifying the coprovisor, we must make each received request context-aware. Each request has its own request control block (RCB), which is used by the coprovisor to set up a DMA context. The design of the RCB is shown in Figure 25, and Table 3 explains the elements of the RCB.

The enhanced version of the coprovisor is shown in Figure 26. It now has two queues, one for buffering requests destined for app1 and the other for buffering requests destined for app2. The scheduling entity is now one block of data buffer (one buffer fragment), rather than an entire request (may consist of blocks of data). The scheduler performs four main functions:

1) Exposing the status of both DMA read and DMA write channels to the device driver (such as if the DMA read channel is idle or in use);

DMA read: Host->FPGA
DMA Write: FPGA->Host
Type 1 interrupt: Raised as the DMA read operation finishes
Type 2 interrupt: Raised as the FPGA finishes processing

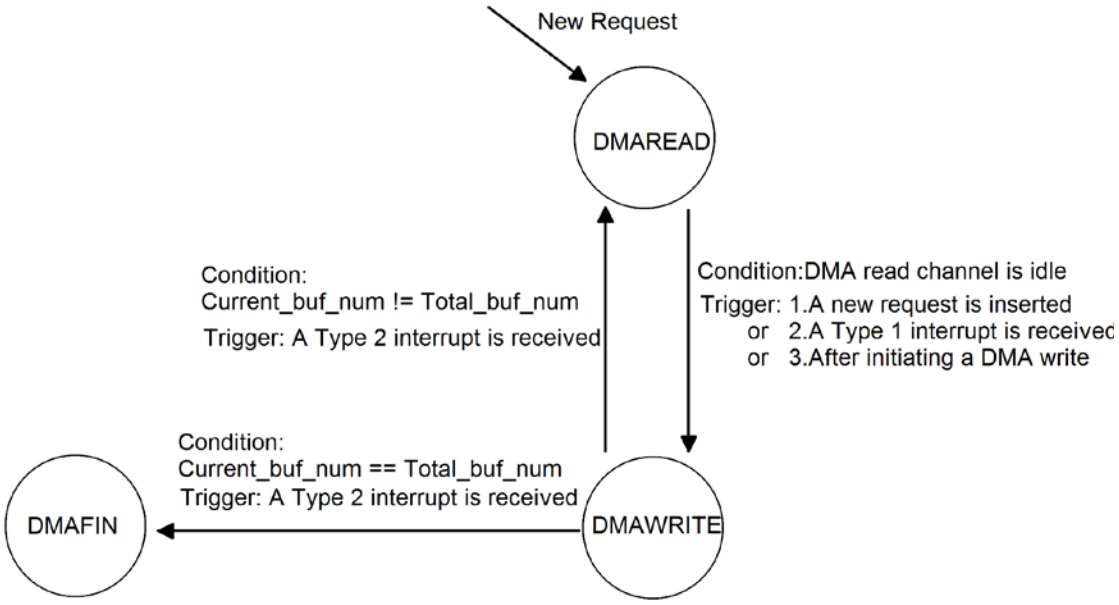


Figure 27. Request state transition diagram

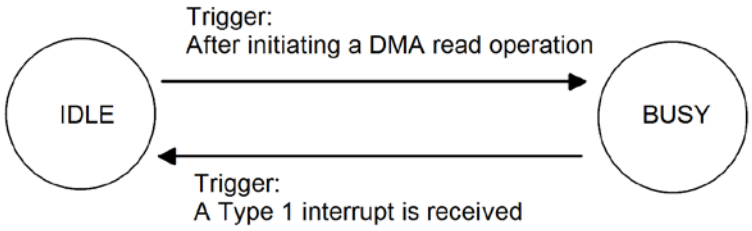


Figure 28. DMA read channel state transition diagram

2) Scheduling a request to use the DMA read channel (this may need to implement a DMA context switch);
3) Maintaining the RCBs of the head requests of the two queues. For example, when one buffer fragment (storing one block of data) of a request is scheduled to use the DMA read channel, the request state will be updated with DMAWRITE, and the "current_buf_num" will be incremented by one;

4) Invoking the Request Remover to remove a request once the request has finished using the FPGA accelerator.

Figure 27 shows the request state transition diagram, and the DMA read channel state transition diagram is shown in Figure 28. When a request is received from a DomU it is marked with DMAREAD state, which indicates that the request is waiting for a DMA read operation. If the DMA read channel is in IDLE state, the DMA read operation will be initiated, the DMA read channel state will be updated to BUSY, and the request state will be updated to DMAWRITE. The Scheduler is invoked in the following three cases: 1) a new request is inserted into a queue; 2) a type 1 interrupt is received; or 3) after initiating a DMA write operation. When the Scheduler is invoked, it will check if the DMA read channel is IDLE and a head request in one of the queues is in DMAREAD state. If so, it will schedule the head request for a DMA read operation.

When a DMA read operation finishes, a Type 1 interrupt will be received from the FPGA accelerator to release the DMA read channel, thereby modifying the DMA read channel state to IDLE. In this case, if the head request in the other queue is waiting for a DMA read operation, the operation can be initiated. When a Type 2 interrupt is received, the context of a request with DMAWRITE state will be loaded into the DMA controller to implement a DMA write operation, and the “Current_buf_num” will be incremented by one. If the head requests of the two queues are both in DMAWRITE state, the Bit 2 of the ASW informs the Scheduler which request has finished a computation on the FPGA.

After initiating a DMA write operation there are two possible cases. One case is that the request has not been completely serviced; that is, there are still data in the DMA buffer associated with the request that have not been processed. In this case, the state of this request will be updated to DMAREAD after initiating the DMA write operation. The second case is that a request has finished all its data processing (Current_buf_num == Total_buf_num). It will be set to the DMAFIN state, and the Request Remover will be invoked to remove the request from the queue.

Chapter 5

Implementation and Evaluation

5.1 Experiments Introduction

Table 4 shows the details of our experimental platform. In the experiments, we first assess the two accelerator designs proposed in Chapter 3. Then in Section 5.3, the virtualization overhead is evaluated with a simple loopback application. In Section 5.4, we verify the validity of pvFPGA with a Fast Fourier Transform (FFT) benchmark. The functionality of the coprovisor is assessed in Section 5.5, through four DomUs sending requests to access the shared FPGA accelerator simultaneously. Finally, we evaluate the improvement in request turnaround time with DMA context switches in Section 5.6.

Table 4. Experimental platform

Name	Description
x86 server	Intel® Xeon Processor W3670 running at 3.2GHZ and 4 GB of main memory
FPGA accelerator	1) Virtex-5 LXT FPGA ML505 Evaluation Kit, 1-lane (Gen 1 [55]) PCIe interface 2) Virtex-6 LXT FPGA ML605 Evaluation Kit, 4-lane (Gen 2 [56]) PCIe interface
VMM	Xen-4.1.2
Native Linux without the Xen VMM	Ubuntu 12.04LTS
Dom0	Ubuntu 12.04LTS
4 DomUs	Ubuntu 10.04LTS, named with Lucid1, Lucid2, Lucid3, Lucid4 separately

5.2 Accelerator Design Evaluation

In this section, we evaluate the two FPGA accelerator designs proposed in Chapter 3 (Figures 15 and 16) with an ML505 evaluation board. We expect the pipelined FPGA (P-FPGA) design to be more efficient than the non-pipelined FPGA (NP-FPGA) design. The ML505 board has a 1-lane PCIe interface, which can be configured to run at 62.5 MHZ with PCIe Generation 1 standard. This evaluation can be directly implemented with the device driver on a native Linux. To measure the performance difference between the two designs, a benchmark based on Xilinx Fast Fourier Transform (FFT) IP core [59] for accelerating 256-Point floating point (single precision) FFT is used. FFT is an algorithm used in a wide range of applications, including signal processing, medical imaging, petrochemical exploration, defense [60]. The DMA read and write operations in figure 16(b) are implemented with 4KB (the page size of Linux OS) data size as a block. Executing this FFT benchmark requires that at least 2KB data (1KB real number and 1KB complex number) is transferred to the FPGA accelerator. Therefore, each block of data has two FFT computations (4KB/2KB = 2).

Implementation with the first design, which has only one DMA channel, is relatively simple. All the data is streamed to the FPGA accelerator by the DMA controller once, and then the results are streamed back to the server memory. Implementation with the second design requires using the pipelined operations. In order to choose a proper pipeline type proposed in Section 3.2, we estimated the FFT benchmark execution latency and the DMA data transfer latency. The FFT benchmark works at 250MHZ. We obtain the FFT benchmark execution latency (about 9.5 us) through a simulation in Modelsim. To measure the communication latency between the server and the FPGA accelerator, we send one block of data (4KB). The server begins time measurement when it initiates the start of the DMA transfer, and once the DMA controller finishes the data transfer, an interrupt is sent to the server. We found that the DMA read and write each take approximately 27 us. Referring to the introduction in Section 3.2, we now have $\mathbf{T_C}$=9.5 us,

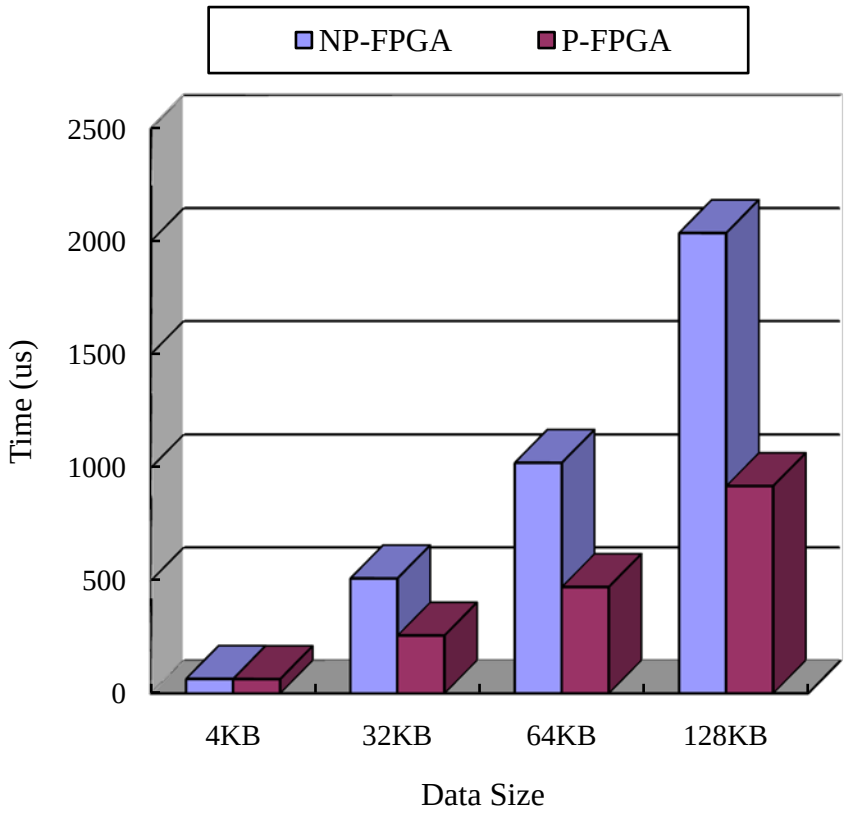


Figure 29. Evaluation of the non-pipelined and pipelined FPGA design

$\mathbf{T_D}$= 27 us. In this case, the DMA data transfer latency is larger than the computation latency ($\mathbf{T_D} > \mathbf{T_C}$), so we can use the pipeline shown in Figure 17(a). As introduced in Section 3.2, the overall turnaround time can be calculated by formula (F-2): $\mathbf{T_D + T_C + N*T_D}$.

We expected the results to match the concluded formulas. For example, when 128KB of data (32 blocks of data) is sent for computations, the turnaround time of NP-FPGA is expected to be (**2*27 + 9.5**) * **32 = 2032** us (formula F-1), and the turnaround time of P-FPGA is expected to be **27 + 9.5+32*27** = **900.5** us (formula F-2).

Figure 29 shows the evaluation results of the NP-FPGA and P-FPGA. The results conform to our concluded formulas. Also, there is no difference between them when sending only one block of data (4KB) for FFT computations on the FPGA accelerator, because no pipelined operations are involved. When more blocks of data is sent for

computations, e.g. 128KB, P-FPGA is over 2 times faster than NP-FPGA.

In the following evaluation and verification experiments, we decided to use the pipelined FPGA design. In order to reduce the communication latency between the server and the FPGA accelerator, we choose to use a new evaluation board (ML605). This board has a 4-lane PCIe interface, which can be configured to run at 250MHZ with PCIe Generation 2 standard.

5.3 Virtualization Overhead Evaluation

In order to estimate the overhead caused by the virtualization layer we added a loopback application, which simply returns any unmodified data it receives, on the FPGA accelerator in the position of app1 in Figure 16(a). Similarly, the DMA read and write operations are implemented with 4KB data size as a block. The FPGA computation latency can be considered zero, since there is no computation of the loopback application. The pipeline of the loopback application is shown in Figure 30. When $\mathbf{T_C}$ equals zero, formula (F-5) can be simplified as:

$$\mathbf{T_D + N * T_D} \qquad \textbf{(F-5)}$$

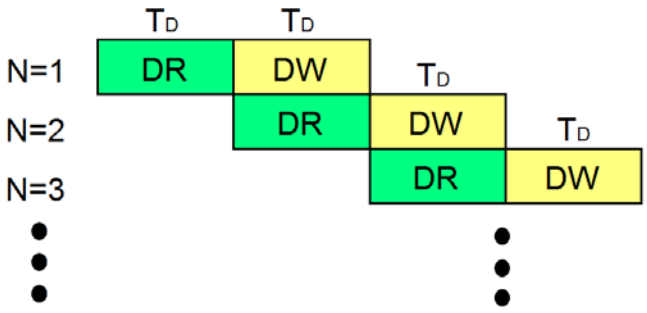


Figure 30. Pipeline of the loopback application

The formula (F-5) shows the turnaround time of the loopback application. When large amounts of data (e.g. **N**=1024) are transferred to the loopback benchmark, we can get an accurate value of $\mathbf{T_D}$, which is the DMA data transfer latency of one block of data.

The evaluation is implemented in the following three contexts:

1) a native Linux environment without the Xen VMM;
2) a virtualized Linux environment but with direct access to the FPGA accelerator (Dom0); and
3) a guest domain in pvFPGA (Lucid1).

Figure 31 shows the time it takes to send data ranging from 32 blocks (128KB) to 1024 blocks (4MB) to the FPGA accelerator for executing the loopback application. The results are averaged across 20 experiments. We found no apparent performance difference between native Linux and Dom0 (driver domain), a result supported by paper [11]. We choose Dom0 as our base case for the following evaluation. When 4MB data (**N**=1024) is sent to the FPGA, the turnaround time is 3586 us. According to formula (F-5), we get $\mathbf{T_D}$ equal to 3.5 us. This value will be used for the following verification experiment.

We summarize the overhead caused by the virtualization layer by comparing the time taken in Dom0 and Lucid1. The overhead time shown in Figure 32(a) varies between 15 us and 30 us with data sizes from 128KB to 4MB. In other words, the overhead time is almost stable with the data size increases. This is because all the data are transferred through shared memory, so the virtualization overhead is not related to the transferred data size. As we described in Section 3.4.1, the only overhead caused by the Xen VMM for the inter-domain data transfer is the two event notifications; one notifying Dom0 that data are ready to send to the FPGA accelerator, and the other notifying the DomU that the results are ready to be fetched. As shown in Figure 32(b), the overhead is close to zero percent compared to the time consumed for sending 4MB data to the FPGA accelerator.

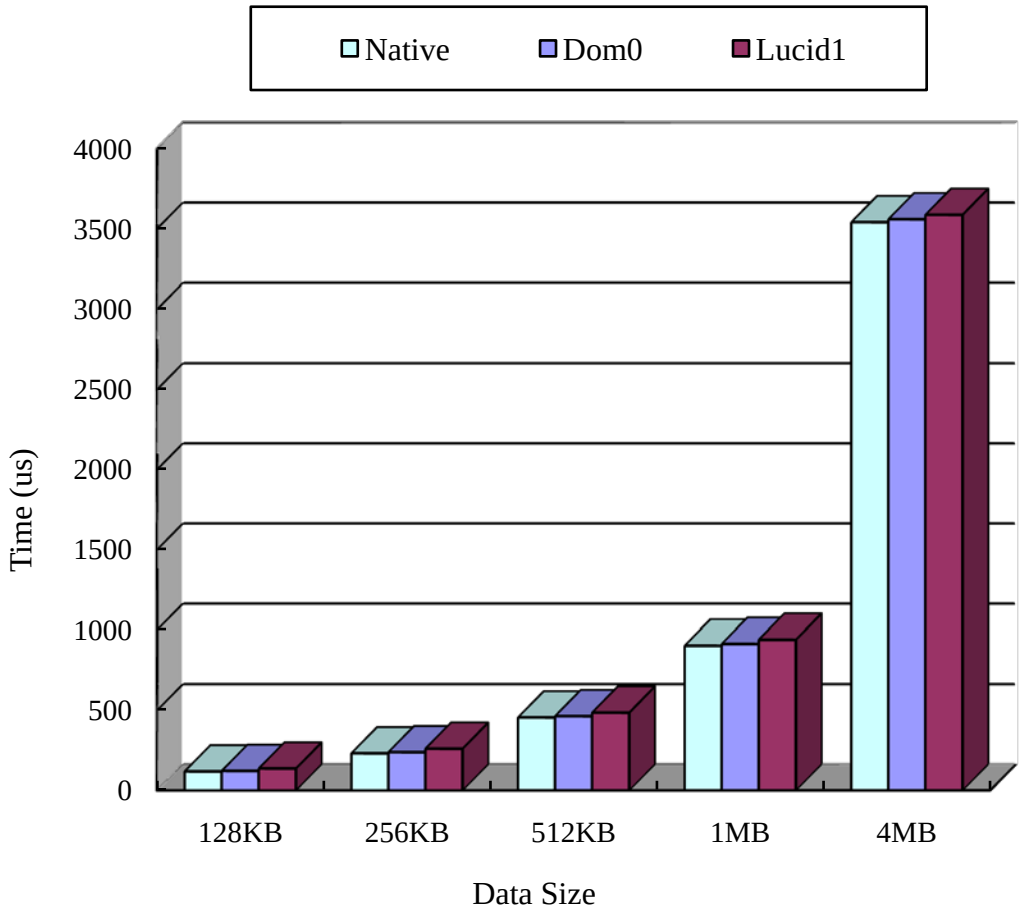
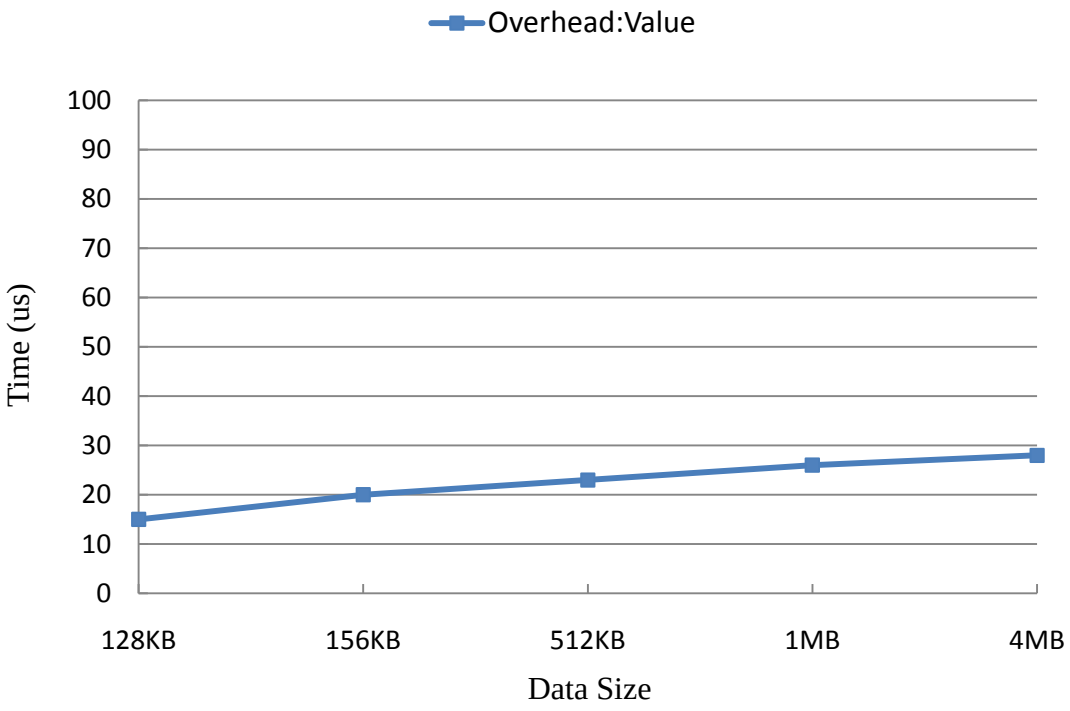
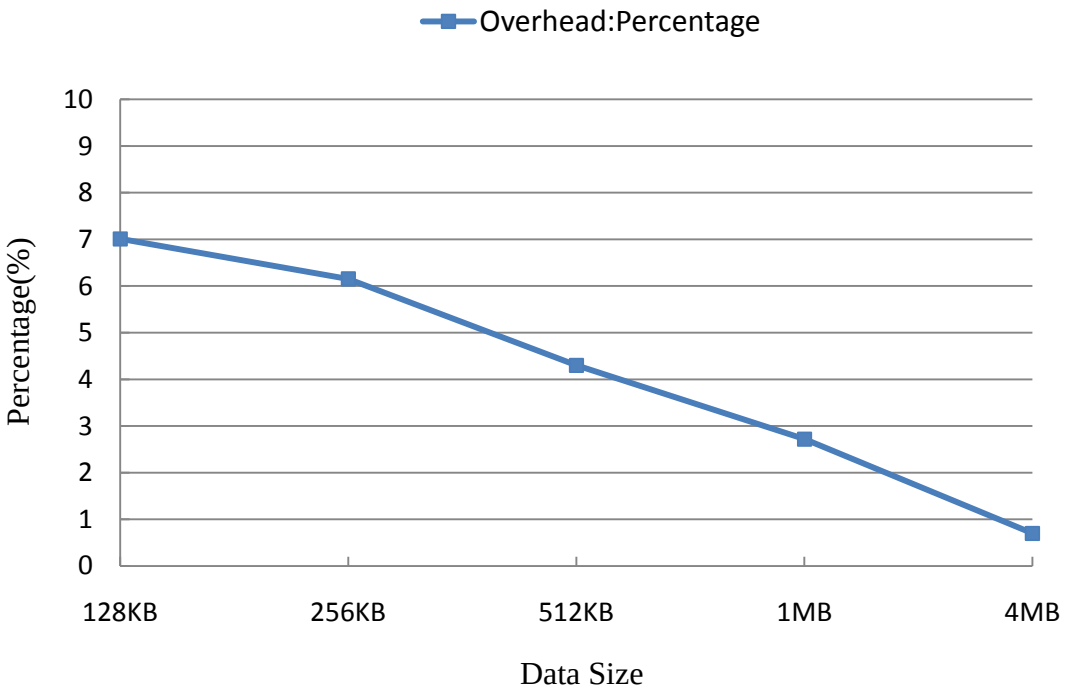


Figure 31. Time for executing a loopback application on the FPGA accelerator



a) Virtualization overhead



b) Virtualization overhead compared to total execution time

Figure 32. Overhead evaluation with a loopback application

5.4 Verification with Fast Fourier Transform

The aim of this test is to verify pvFPGA for accelerating a real application. We built a benchmark based on Xilinx FFT IP core for accelerating 256-Point floating point (single precision) FFT in the first evaluation. Here, we change the loopback app with the FFT benchmark; the FFT benchmark module still operates at 250MHz. In the experiments we send 128KB to 4MB of floating point data to compute FFT on the FPGA accelerator. Data is still transferred in a pipelined way, with 4KB as a block. $\mathbf{T_C}$ still equals 9.5 us which is the same as we got in the first evaluation with Modelsim. As we obtained through the experiment in Section 5.3, $\mathbf{T_D}$ equals 3.5 us. We expected the experimental results would conform to the formula (F-4). For example, when 32 blocks of data are sent for FFT computations, the turnaround time is expected to be **3.5 + 32*(9.5 + 3.5) = 419.5** us.

We verified with Matlab that all the FFT results received from the FPGA accelerator in a DomU application are accurate. Figure 33 shows the turnaround time that is required for computing FFT with different data sizes on a CPU, and with the FPGA accelerator. The results are in accordance with formula (F-4); for example, when 32 blocks of data are sent for FFT computations on the FPGA accelerator, the experimental turnaround time is 436 us, which is close to the result from the formula (419.5 us). We sent requests to the FPGA accelerator to compute FFT from both Dom0 and a DomU (Lucid1) to verify the overhead. As shown in Figures 34(a) and Figure 34(b), the overhead is close to zero percent when 4MB or more data is required for computations.

The baseline code we chose to run on a CPU was FFTW3. FFTW3, a free collection of fast C routines, is commonly used by researchers to computing FFT [61]. Using FFTW3 requires an initialization at the start, which creates a plan for the array that will be used for computing FFT. The array can then be used repeatedly if the user needs to compute FFT multiple times. As shown in Figure 31, the speedup is over 2 fold when 256KB (or less) data is required for FFT computations. The influence of the initialization overhead is minimized as the data size increases. The speedup remains at approximately

1.1 fold with 4MB data. Here, the FPGA computation takes most of the overall time; for example, when 4MB data is sent for FFT computations, the FPGA computation time is 9.5us*1024=9728 us, which is about 73% of the overall time. To gain more speedup requires either further optimization of the FFT computation core on the FPGA accelerator or using a faster communication channel (such as a PCIe interface with more lanes), so that the communication overhead between the FPGA accelerator and the server is reduced.

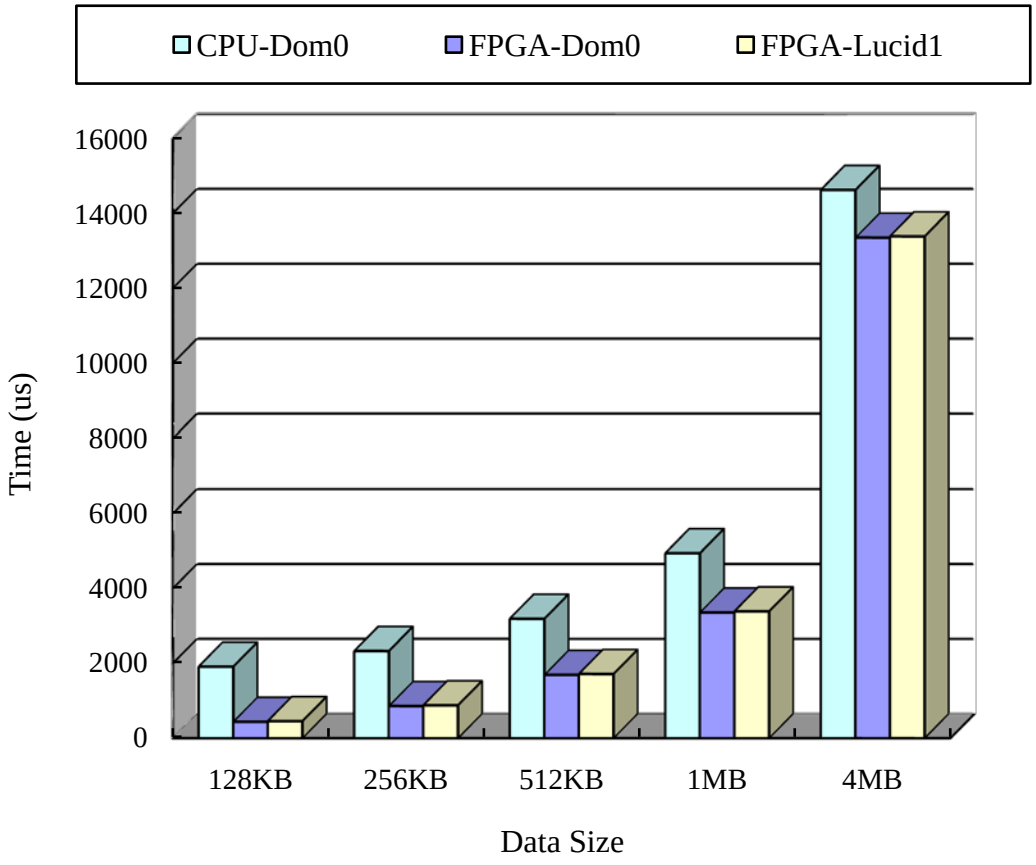
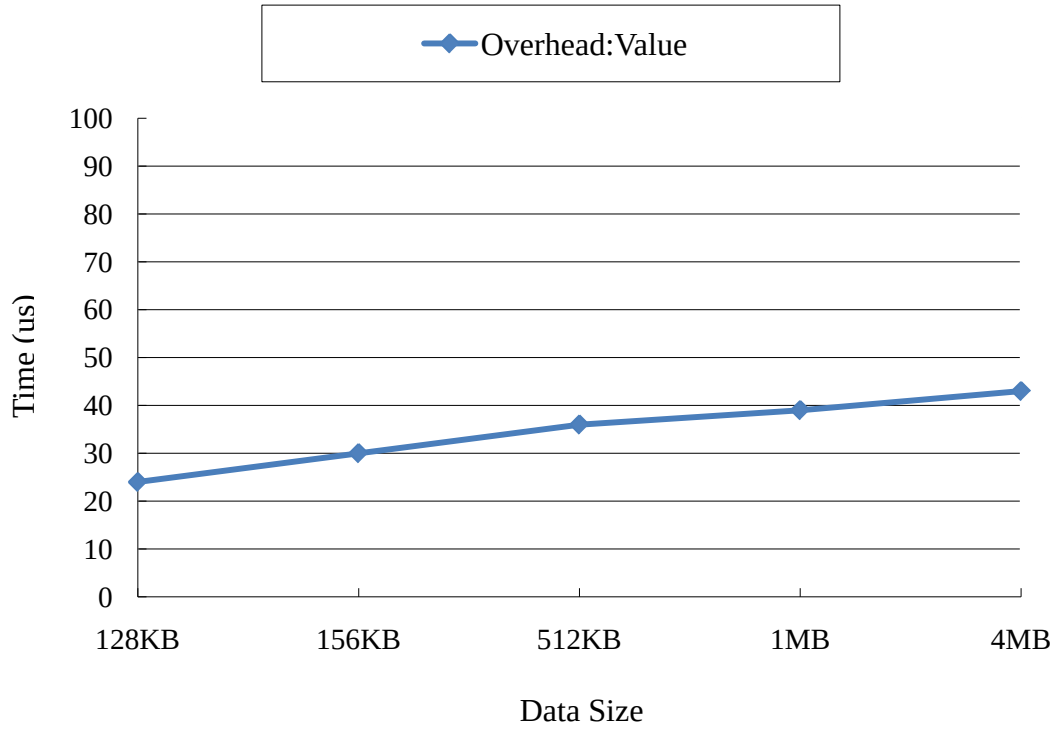
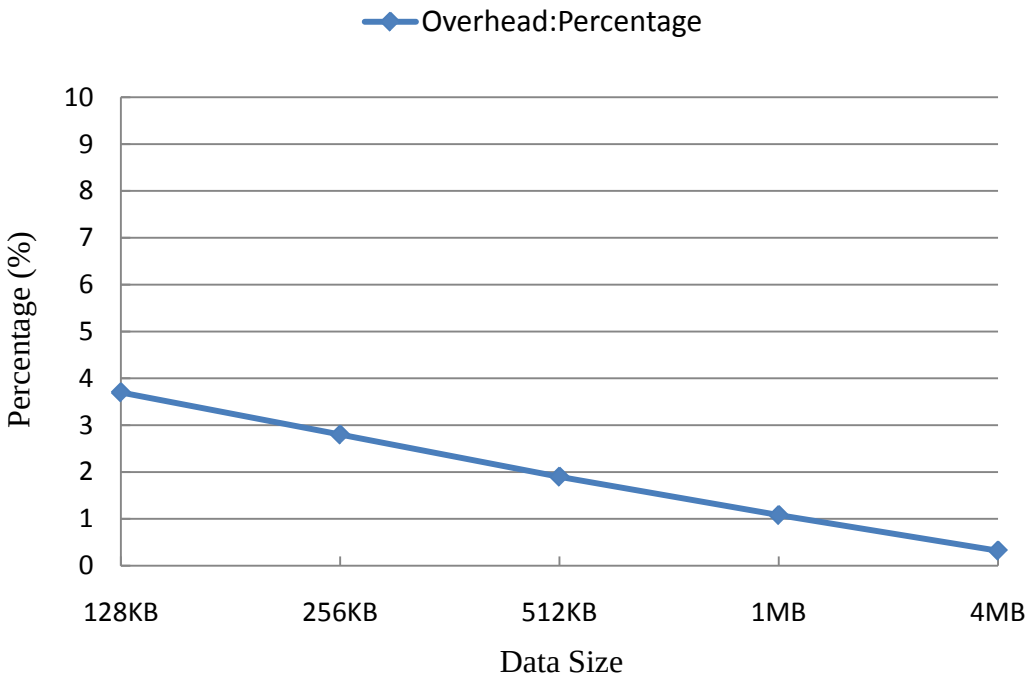


Figure 33. Verification with an FFT application



a) Virtualization overhead



b) Virtualization overhead compared to total execution time

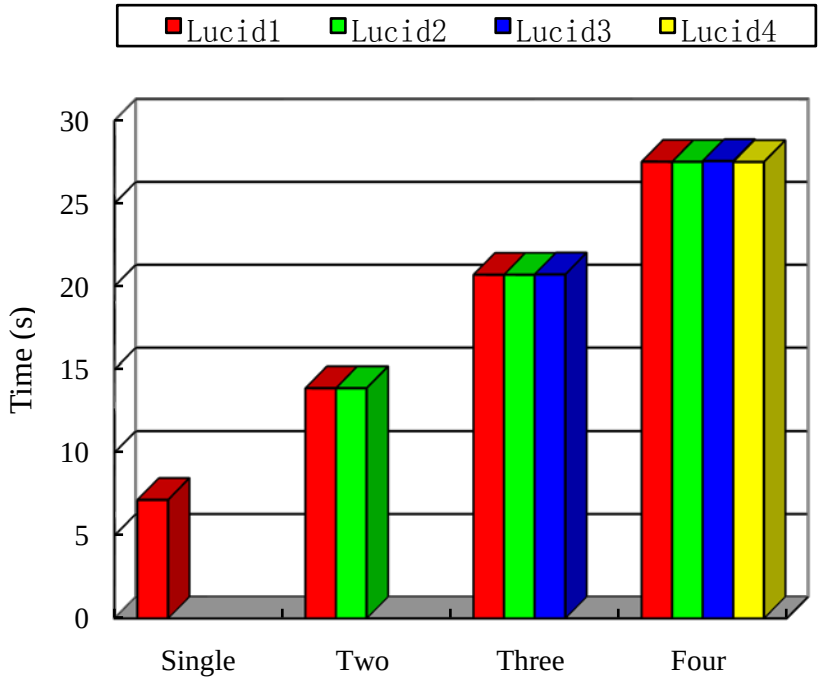
Figure 34. Overhead evaluation with an FFT application

5.5 Coprovisor Evaluation

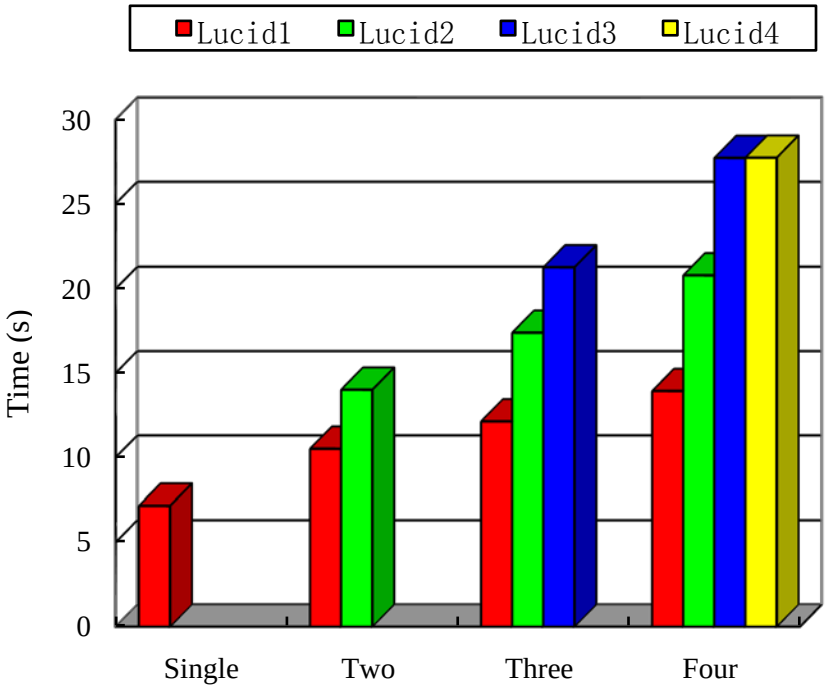
We next evaluate the functionality of the coprovisor, which multiplexes multiple DomUs to access the FPGA accelerator. Each of the four DomUs (Lucid1, Lucid2, Lucid3 and Lucid4) allocates a 1MB size data pool, which is shared with Dom0 for inter-domain data transfer. We still use the FFT benchmark for the evaluation, so we set all the applications in the four DomUs to select app1 when they request services, and the data size for all the requests is set to 1MB via the command channel. The evaluation task is for all of them to send 2GB of data to compute FFT on the FPGA accelerator, so each DomU sends 2048 requests to complete the computation.

The turnaround time of the four DomU simultaneously accessing the FPGA accelerator to compute FFT with 2GB of data is shown in Figure 35(a). When a DomU accesses the FPGA accelerator individually, the total time for finishing FFT with 2GB of data is approximately 7 seconds, whereas when all four DomUs contend for one FPGA accelerator it takes about 27.5 seconds.

To distribute different maximum data transfer bandwidths to DomUs in pvFPGA, we assign different sized data pools in each DomU. In this experiment, we assigned Lucid1, Lucid2, Lucid3 and Lucid4 with 1MB, 512KB, 256KB and 256KB data pool sizes respectively, separately in their kernel spaces. Lucid1 is treated as a VIP domain, as it is assigned the largest data pool size. Figure 35(b) shows the turnaround time of finishing the evaluation task with different data pool sizes in the DomUs. When all four DomUs request access to one FPGA accelerator simultaneously, Lucid1 gets serviced twice as fast as Lucid3, and Lucid3 and Lucid4 spend equal time (approximately 27.7 seconds) to get their requests serviced, due to their equal assigned data pool size.



a) Identical data pool size in DomUs



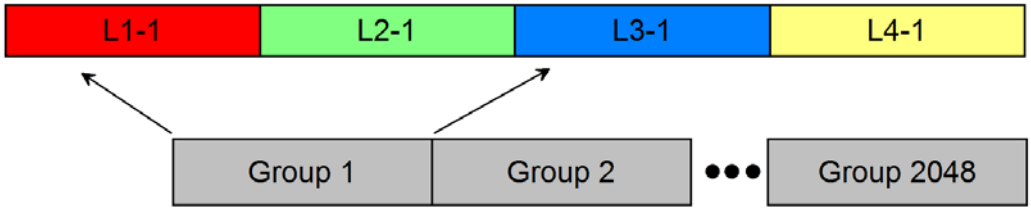
b) Different data pool sizes in DomUs

Figure 35. Evaluation of the coprovisor

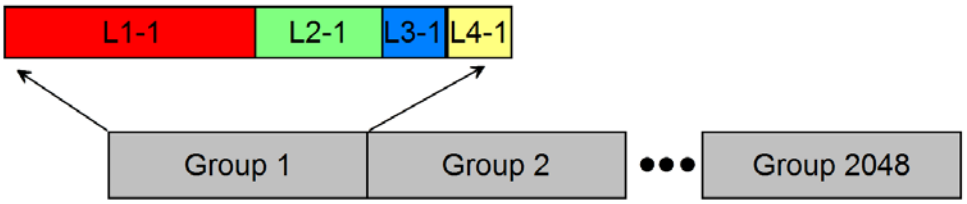
5.5.1 Contention Analysis

We first consider the situation in Figure 35(a) where the four DomUs contend for access to the FPGA accelerator with equal sized data pools. Figure 36(a) shows how these requests are scheduled. L1-1 represents the first request from Lucid1, L2-1 represents the first request from Lucid2, and so on. In Figure 35(a), each DomU is assigned with a 1MB data pool, so each domain needs to send 2048 requests to finish 2GB data of FFT computations. A DomU cannot send the second request until the first request is serviced, because each DomU has only one data pool. Thus, all the requests can be grouped into 2048 groups, with each group having a request from each of the DomUs. Here, every request has 256 blocks of data (1MB/4KB); that is, **N** equals 256. After sending 2048 groups of requests, all the DomUs will complete their tasks. They finish with almost the same turnaround time, which is also the turnaround time of the 2048 groups of requests. One group of requests has four requests from each of the four DomUs. According to formula (F-4), the turnaround time of one group of requests, $\mathbf{T_g}$, is [3.5+256*(9.5+3.5)]*4 = 13326 us. Thus, the turnaround time of 2048 groups of requests is 2048* $\mathbf{T_g}$ = 27.3 s, which means each of the four DomUs needs 27.3 s to get their requests serviced. Before sending a request, each DomU needs to load data that needs computation into the data pool; this can be viewed as preprocessing work. However, it is not necessary to add the preprocessing time to the calculated result. In Figure 36(a). When the first request from Lucid1 (L1-1) finishes, it will load new data to the data pool for the second request (L1-2 in Group 2). Meanwhile, the first request from Lucid2 (L2-1) is scheduled for computation. The preprocessing time of L1-2 overlaps the time for servicing L2-1, so the preprocessing time is hidden. The experimental result in Figure 35(a) is 27.5 us which is very close to the concluded result.

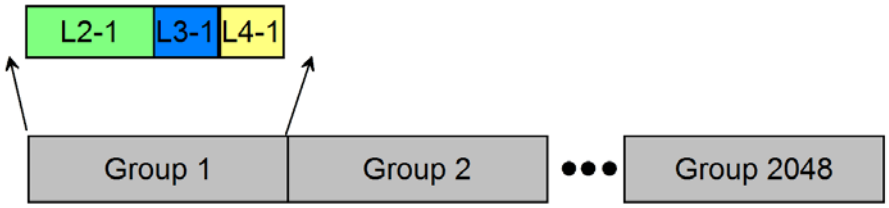
We next consider the situation in Figure 35(b), where the four DomUs that are assigned different data pool sizes contend for access to the shared FPGA accelerator. The contention analysis is more complex, since VIP domains will finish their requests first and



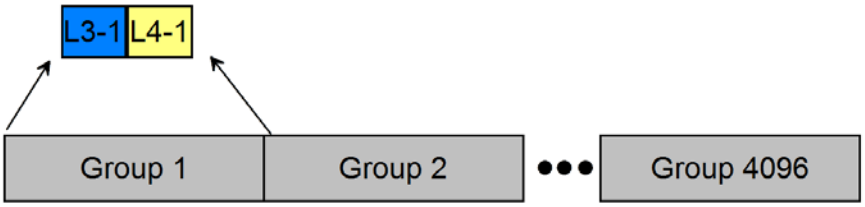
a) Contention analysis of Figure 35(a)



b) Phase1 contention analysis of Figure 35(b)



c) Phase2 contention analysis of Figure 35(b)



d) Phase3 contention analysis of Figure 35(b)

Figure 36. Analysis of four DomUs contending for access to the shared FPGA accelerator

are out of the contention at a certain time instance. Here, we split the contention into three phases. In each phase a DomU completes its evaluation task, and will be out of contention for access to the shared FPGA accelerator. Phase1 has Lucid1, Lucid2, Lucid3 and Lucid4 contending for the FPGA accelerator, while in Phase2 only Lucid2, Lucid3 and Lucid4

contend for the accelerator, since Lucid1 has finished its requests and is out of contention at the end of Phase1. In Phase3, Lucid3 and Lucid4 contend for the FPGA accelerator, as Lucid2 has finished its requests and is out of contention at the end of Phase2.

Phase1: Lucid1, Lucid2, Lucid3 and Lucid4 contend for the shared FPGA accelerator. Lucid1 will finish all its requests at the end of Phase1. Figure 36(b) shows how the requests are scheduled, which is slightly different than in Figure 36(a). L1-1 is twice as long as L2-1 and four times longer than L3-1, because Lucid1 is assigned a 1MB data pool, while Lucid2 is assigned a 512KB data pool and Lucid3 a 256KB data pool. After 2048 groups of requests, phase1 ends because Lucid1 has completed its task (1MB*2048= 2GB). After Phase1, Lucid2, Lucid3 and Lucid4 have only partially finished their tasks. Table 5 shows the remaining data size of each DomU at the end of Phase1.

Table 5. Remaining data size of each DomU at the end Phase1

DomU	Remaining data size
Lucid1	2GB-1MB*2028 = 0
Lucid2	2GB-512KB*2048 = 1GB
Lucid3	2GB-256KB*2048 = 1.5GB
Lucid4	2GB-256KB*2048 = 1.5GB

In Phase1, the turnaround time of one group of requests, $\mathbf{T_g}$, is equal to the sum of the request turnaround time of the four DomUs. Therefore:

$\mathbf{T_g}$ = 3.5+256*(9.5+3.5) + 3.5+128*(9.5+3.5) + 3.5+64*(9.5+3.5) + 3.5+64*(9.5+3.5)
= 6670 us.

The turnaround time of Lucid1 equals the **13.7 s** turnaround time of Phase1 (2048* $\mathbf{T_g}$). As shown in Figure 35(b), our experimental result of 13.9 s is close to the calculated result.

Phase2: As shown in Figure 36(c), in this phase Lucid2, Lucid3 and Lucid4, contend for access to the shared FPGA accelerator. Table 5 shows that Lucid2 has 1GB data left for FFT computations, so the number of groups in Phase2 is 2048 (1GB/512KB). Table 6

Table 6. Remaining data size of each DomU at the end Phase2

DomU	Remaining data size
Lucid2	1GB-512KB*2048 = 0
Lucid3	1.5GB-256KB*2048 = 1GB
Lucid4	1.5GB-256KB*2048 = 1GB

shows the remaining data size of each DomU at the end of Phase2.

In Phase2, the turnaround time of one group of requests, $\mathbf{T_g}$, equals the sum of the request turnaround time of Lucid2, Lucid3 and Lucid4. Therefore:

$\mathbf{T_g}$ = 3.5+128*(9.5+3.5) + 3.5+64*(9.5+3.5) + 3.5+64*(9.5+3.5) = 3338.5 us .

The turnaround time of Lucid2 is equal to the turnaround time of Phase1 and Phase2, which is 13.7 s + 2048* 3338.5us = **20.5 s**. our experimental result is 20.8 s which is close to the concluded result.

Phase3: As shown in Figure 36(d), only Lucid3 and Lucid4 contend for access to the shared FPGA accelerator. So we get $\mathbf{T_g}$ = 3.5+64*(9.5+3.5) + 3.5+64*(9.5+3.5) = 1671 us.

Both Lucid3 and Lucid4 have 1GB data left for computations, so after 4096 groups (1GB/256KB) of requests are scheduled Lucid3 and Lucid4 have completed their tasks. Therefore, the turnaround time of Lucid3 equals the turnaround time of Lucid4, which is also the turnaround time of the three phases. This is calculated as: 20.5 s + 4096*1671 us = **27.3 s**. Our experimental result is 27.7 s, which is close to the calculated result.

As shown, the overall turnaround time (Phase1+Phase2+Phase3) of Figure 35(b) equals that of Figure 35(a), since their overall data sizes are the same (4*2GB = 8GB). In Figure 35(a), all the DomUs get their requests serviced at an equivalent and invariable speed. In Figure 35(b), Lucid3 and Lucid4 get their requests serviced at a slower speed at the beginning, but as Lucid1 and Lucid2 drop out of the contention this speed increases.

5.6 DMA Context Switching Evaluation

In this section, we evaluate the improvement of request turnaround time due to the use of DMA context switches. In Chapter 4, we proposed the improved design of pvFPGA. The aim of the new design is to eliminate the scheduling delay through DMA context switches when two requests bid to use different apps on the FPGA accelerator, thereby reducing the request turnaround time. All the above evaluations are implemented with requests requesting to use the same app on the FPGA accelerator. Here, we set requests from Lucid1 and Lucid2 to access app1, and requests from Lucid3 and Lucid4 to access app2.

At the FPGA accelerator end, we used a design that emulates the computation procedure. As shown in Figure 37, both app1 and app2 have a buffer for storing a block of incoming data, and a timer for emulating algorithm computation time; the timer asserts an interrupt request when the time is up. The previous experiments are all implemented with an app whose computation latency is at the microseconds level. In order to show that our proposed accelerator design can be used for general purpose computing, regardless of the app computation latency, we set the app1 timer to 4 seconds and the app2 timer to 2 seconds for one block of data (4KB). In this case, app1 and app2 emulate two algorithm computations that require 4 seconds and 2 seconds respectively to complete one block of data processing.

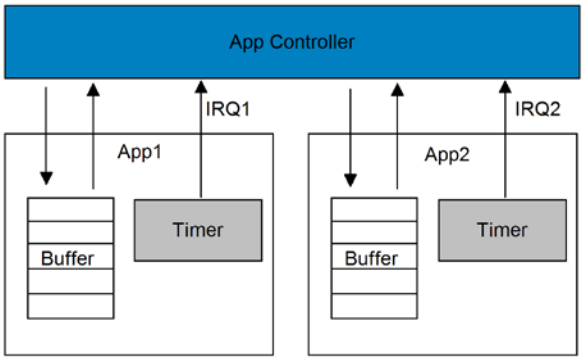


Figure 37. An accelerator emulating design for verification purposes

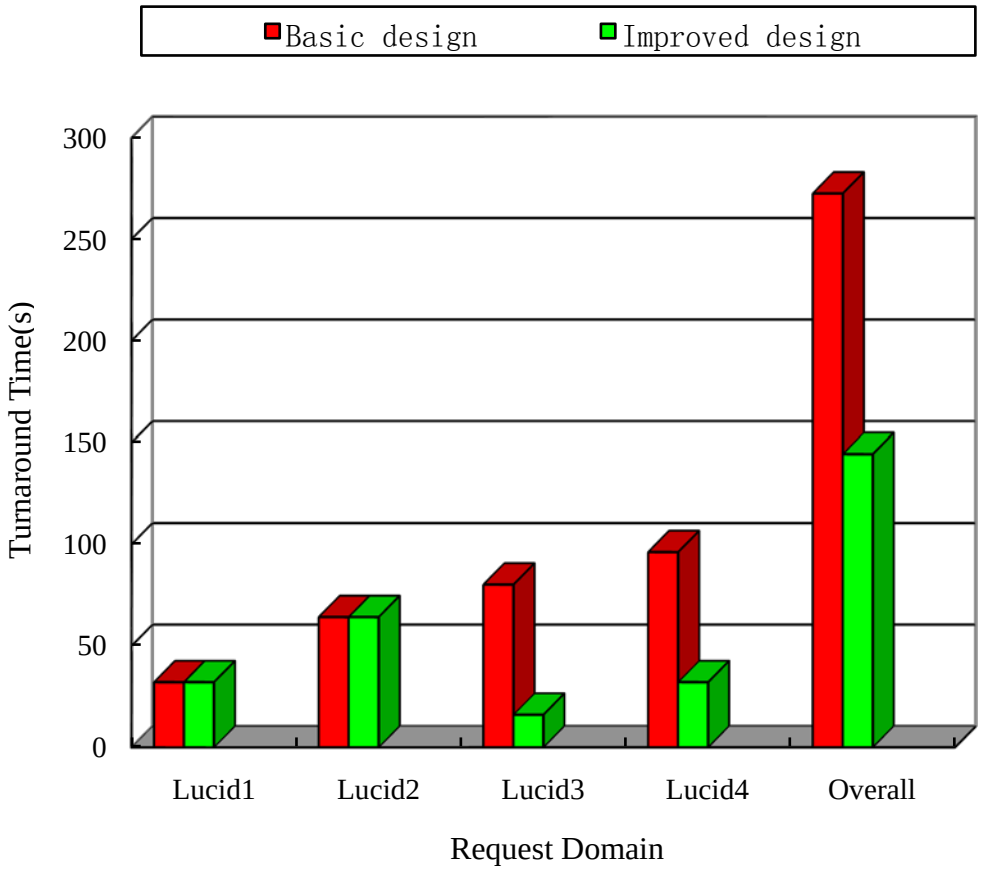


Figure 38. Request turnaround time comparison between the basic design and improved design

Figure 38 shows the turnaround time when each DomU sends a request with eight blocks of data (32KB) to the FPGA accelerator simultaneously. The improved design does not affect Lucid2 because its request is inserted into the same queue as Lucid1's request, and requests in the same queue are scheduled in FCFS order. The most apparent improvement (an 80% reduction of the basic design's turnaround time) is when the Lucid3 request is inserted into a different queue than the Lucid1 request. DMA context switching helps reduce the scheduling delay caused by scheduling the requests from Lucid1 and Lucid2. The concurrent share of the DMA read channel enables simultaneous use of app1 and app2 on the FPGA accelerator. With the improved design, the turnaround time of all the requests is reduced to approximately 53%.

Chapter 6

Related Work

6.1 FPGA Virtualization

6.1.1 Design Review

Paper [18] describes the state-of-the-art FPGA virtualization solution. The authors have proposed a solution for virtualizing an FPGA accelerator for multiple processes. An important component in their virtualization work is what they refer to as a “Virtual Coprocessor Monitor (VCM)”, which multiplexes requests from multiple processes to access the shared FPGA accelerator.

The work presented in [18] does not include the design of an FPGA accelerator, and the FPGA design they use is vendor proprietary. Therefore, their proposed VCM, which is designed as a user-level application, interacts with the FPGA accelerator driver through vendor-supplied APIs. They modify the vendor-supplied APIs with virtualization APIs, which can be used by user processes to interact with the VCM.

The VCM in [18] is made up of three components: a virtualization manager, a request queue and virtual memory space. The virtualization manager implements time-sharing mechanisms. When a user process sends a request to the VCM to access the FPGA accelerator, the VCM creates a virtual memory space. It shares this space with the calling process through conventional POSIX memory sharing IPC (Inter-Process Communication) primitives. Then the user process copies data to the virtual memory space, followed by the insertion of a request into the request queue. When the FPGA accelerator has finished, the VCM will sends an acknowledge signal to the user process indicating that results are ready

to be retrieved. The VCM also provides an API for a process to release the virtual memory space.

6.1.2 Comparison with pvFPGA

The work presented in paper [18] has successfully virtualized an FPGA accelerator for multiple processes from single OS, whereas pvFPGA has successfully virtualized an FPGA accelerator for multiple domains (or virtual machines). Our design proposes a component called coprovisor for multiplexing requests from multiple domains for access to an FPGA accelerator, while in [18] they proposes a VCM for multiplexing multiple processes to access an FPGA accelerator. Both the coprovisor and VCM use the FCFS scheduling algorithm. However, the difference is that the VCM runs as a user space application, while the coprovisor sits directly on the device driver layer. Evaluation results have shown that their virtualization overhead reaches approximately 23%; the virtualization overhead in pvFPGA is close to zero.

6.2 Inter-domain Network Virtualization

6.2.1 Design Review

Several papers [53-56] have proposed solutions for enhancing the performance of network packets passed between virtual machines residing in the same physical machine. Though these works are not related to virtualization for high performance computing (HPC) or accelerators, their designs have something in common with pvFPGA and the recent GPU virtualization solutions. For example, all use shared memory for inter-domain data transfer, thereby achieving better performance than with other methods. We review the state-of-the-art inter-domain network virtualization solution, XenLoop [55], in this section.

We typically expect TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) based network communication performance between two processes in their own domains that co-reside in the same physical machine to outperform processes that reside in different physical machines. However, papers [53-56] show that the inter-domain network communication has lower performance which is unexpected.

The lower performance is mainly caused by the TCP/IP processing cost, as well as the Xen page flipping mechanism for inter-domain data transfer. The data packets must be directed to Dom0, which then passes them to the destined domain. As described in paper [62], the page flipping mechanism transmits data by transferring ownership of the machine page. The Xen VMM provides a hypercall to achieve the machine page ownership swapping. When large volumes of network packets need to be transferred, the repeated issuance of hypercalls results in high overhead. Also, the translation look-aside buffers (TLBs) need to be flushed after each flipping, because the translation of virtual address to physical address is modified.

XenLoop is a kernel module, which works beneath the network layer. It achieves full transparency for user-level programs; that is, user programs or libraries do not need to be rewritten using XenLoop. Every outgoing packet is intercepted by XenLoop, with the help of netfilter [63]. The destination address of a packet is resolved to a layer 2 Media Access Control (MAC) address with the support of the ARP (Address Resolution Protocol) cache. The XenLoop module also maintains a table that stores [DomainID, MAC address] pairs. The MAC addresses of the domains co-residing in the same physical machine are stored in this table. Using the table, a resolved MAC address determines if the packet is destined to a domain residing in the same physical machine. Figure 39 shows an example of ARP cache content generated from Dom0. In the figure, the four IP addresses are those assigned to four DomUs running over the Xen VMM. Table 7 shows the corresponding XenLoop tables stored in a DomU with ID of 1. In this example, any packet with a destination address in Table 7 will be forwarded to the related shared buffer; otherwise,

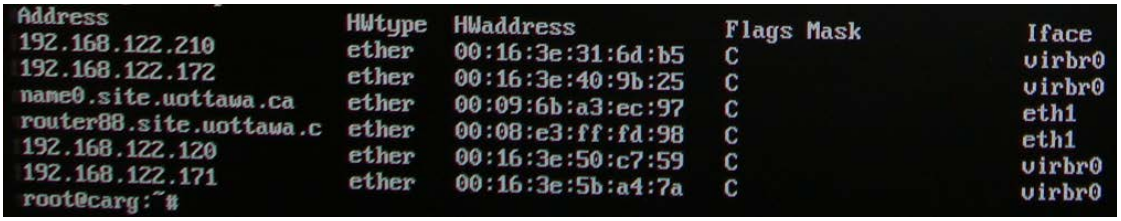


Figure 39. An example of ARP cache content generated from Dom0

Table 7. An example of XenLoop table stored in a DomU

Domain ID	MAC address
2	00:16:3e:50:c7:59
3	00:16:3e:40:9b:25
4	00:16:3e:31:6d:b5

the packet will be forwarded to Dom0.

The following describes these two cases:

1) When a TCP/IP packet is sent to a domain outside its physical machine, it will be directed to the split device driver, and then sent to the network card driver. This does not use XenLoop.
2) When a TCP/IP packet is sent to a domain residing in the same physical machine, the packet will be sent through the XenLoop inter-domain communication channel. The packets are passed between the sender and receiver domains directly, whereas the original method was to redirect the packets through Dom0.

The XenLoop inter-domain communication channel includes two FIFO buffers (one for sending and one for receiving packets) shared between two domains, and an event channel for sending asynchronous notifications. Their experiments show that using XenLoop the bandwidth is increased by a factor of six compared to using the original communication mechanism.

6.2.2 Comparison with pvFPGA

Both XenLoop and pvFPGA use shared memory for inter-domain data transfer, which results in low overhead. The difference is that XenLoop is specifically designed for intercepting and redirecting TCP/IP packets, while pvFPGA uses shared memory to transfer data to an FPGA accelerator without redirection. Also, XenLoop does not include a device driver design, because no data is transferred to any underlying hardware unit. In terms of data transfer, XenLoop uses a 2-copy operation; that is, TCP/IP packets need to be copied to the shared FIFO buffer from the sender's network buffer, and at the receiver end the packets must be copied to the receiver's buffer. There is no such a copy operation in pvFPGA, since the inter-domain shared buffer is mapped to a user application address space, and the user's data can be stored in the buffer directly.

6.3 GPU Virtualization

6.3.1 Design Review

By now, many papers have successfully achieved GPU virtualization. In this section, we review some current GPU virtualization solutions.

1) GViM

GViM is a Xen-based system design solution that permits users to run any CUDA-based applications in a VM [11]. It uses the shared memory mechanism for data transfer between VMs, and an interposer library in each guest VM provides CUDA accesses. CUDA calls from user applications in a VM are intercepted, packed together with parameters into CUDA call packets by the interposer library, and then passed to the frontend driver, which transfers them to the Dom0 backend driver. In Dom0, the CUDA call packets are continuously passed to the library wrapper, which converts them into

standard CUDA function calls. Each guest VM has a dedicated CUDA call buffer in Dom0. CUDA calls from guest VMs requesting access to the GPU are stored in the buffer first, the buffered requests are then scheduled to be translated into CUDA function calls in a round-robin fashion.

2) vCUDA

In some respects, vCUDA functions much the same as GViM; for example, the latest version of vCUDA also uses shared memory for data transfer between VMs. CUDA calls in a guest VM are intercepted and packed by a vCUDA library. The vCUDA library has a vGPU component that reveals the device information (e.g. GPU memory usage, texture memory properties) to applications in a guest VM. To ensure information consistency, vCUDA includes a synchronization mechanism between the vGPU and a component in Dom0 known as vCUDA stub. The vCUDA library has a global queue for storing all the packed packets, which are periodically transferred to the vCUDA stub. The vCUDA stub unpacks the received packets and invokes the related CUDA API calls in Dom0. The designers also propose using a Lazy RPC (Remote Procedure Call) mechanism to batch specific RPCs, thereby reducing the number of expensive world switches (context switches between different domains). vCUDA uses working/service threads (introduced in [14]) to enable requests to be concurrently executed on the GPU accelerator, regardless of whether they come from the same or different domains.

3) gVirtuS

gVirtuS [10] is a VMM independent solution for GPU virtualization in a cluster environment. Intercepted CUDA calls in a guest VM are redirected to the host domain running on a different physical machine via a TCP/IP based communicator. A resource sharing framework was proposed as an extension of gVirtuS in [13]. The authors created a virtual process context to consolidate different applications (including from different domains) into a single application context, in order to time share or space share streaming multiprocessors (SMs) in a GPU accelerator.

6.3.2 Comparison with pvFPGA

Comparing pvFPGA with GPU virtualization solution might not seem fair, because GPUs are designed for general purpose computing. Here, we compare them only in terms of the virtualization techniques being used (see Table 8 for a summary of the comparison results). Though both methods use shared memory for data transfer between VMs, the solutions of intercepting user space API calls from frontend domains and redirecting them to the backend domain are specific to GPUs. Also, due to the limited knowledge of GPU hardware specifications, it is not feasible to achieve GPU virtualization at the low device driver layer, which can provide higher efficiency and lower overhead. A CUDA application might entail calling thousands of CUDA APIs [14], whereas only one "call" is required to access the FPGA accelerator in pvFPGA. The overhead on GPU virtualization solutions average 11%, whereas pvFPGA overhead is near zero when large volumes of data are transferred. Moreover, no GPU virtualization solutions include a GPU multiplexing scheme to supply DomUs with different maximum data transfer bandwidths. The coprovisor efficiently multiplex requests to access the FPGA accelerator at the device driver layer, and pvFPGA can provide different maximum data transfer bandwidths for DomUs by regulating the size of the shared data pools.

Table 8. Comparison of pvFPGA with the current GPU virtualization solutions

Solution	GViM	vCUDA	gVirtuS	pvFPGA
VMM	Xen	KVM	VMM independent	Xen
Service Request Method	API calls redirection	1. API calls redirection 2. vGPU 3. Lazy RPC	API calls redirection	"Write()" system call
Parameter Passing Method	Packed together with API calls and redirected to the privileged domain	Packed together with API calls and redirected to the privileged domain	Packed together with API calls and redirected to the privileged domain	Shared Memory
Multiplexing Method	Resource Management Logic with round-robin scheduling algorithm	Work/Service thread with FIFO scheduling algorithm	A virtual process context consolidating different applications into a single application context to time share or space share SMs on a GPU	Coprovisor with FIFO scheduling algorithm
Virtualization Overhead	6% - 25%	11% on average	Depends on the selected VMM	Close to zero

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we proposed pvFPGA which is the first system design solution for virtualizing FPGA-based hardware accelerators on x86 servers. In pvFPGA, a frontend driver in a DomU kernel space allocates a data pool. A user space process can map the data pool into its address space, so it can access the data pool directly. The data pool is mapped to the Dom0 kernel space through the grant table mechanism provided by the Xen VMM, so the FPGA accelerator driver can also access the shared data pool directly. In this way, the shared data pool is used for both user-kernel and inter-domain data transfer. Our experiments have demonstrated that the virtualization overhead in this solution is near zero, because the only overhead caused by the Xen VMM is the two event notifications, as illustrated in Section 3.4.1.

We also proposed a component that we term a coprovisor, which is responsible for multiplexing requests from multiple guest domains to access the shared FPGA accelerator. The coprovisor sits directly on the device driver layer in Dom0. Requests from DomUs are first buffered in a queue, and then scheduled to the FPGA accelerator in an FCFS order. pvFPGA provides DomUs with different “Qualities” of service, by assigning different data pool sizes in DomUs. Our experiments have shown that a “VIP” domain (with a larger data pool) has a higher data transfer bandwidth, so it gets serviced faster than an ordinary domain (with a smaller data pool) when they are contending for the shared FPGA accelerator simultaneously.

In addition, we further improved pvFPGA by including a capability to implement DMA context switches, which reduce the scheduling delay. Our experimental results have

shown that the improved design with DMA context switches reduces the request turnaround time.

In the final section of the thesis, we compared our pvFPGA solution with the state-of-the-art FPGA, network and GPU virtualization solutions. The start-of-the-art FPGA virtualization solution is still at the multitasking level on a single OS. The latest network packets transfer virtualization solution is specifically designed to intercept and redirect TCP/IP packets. And the prevailing GPU virtualization solutions are specific to virtualizing GPU accelerators, which are intended to intercept and redirect API calls to access GPUs. By comparison, pvFPGA is specific to virtualizing an FPGA accelerator for multiple domains or virtual machines, and it operates directly at the device driver layer. Our experimental results have shown that pvFPGA has lower virtualization overhead (close to zero) than the current GPU virtualization solutions (11% on average).

7.2 Future Work

Here we discuss possible future extensions of our current pvFPGA design.

1) Partial Reconfiguration

As introduced in Section 2.1.3, partial reconfiguration enables partial modules in the FPGA to be reconfigured while other modules run without interruption. pvFPGA can take advantage of partial reconfiguration to satisfy more diverse demands from DomUs. More specifically, DomUs can have their own applications in the form of partial bitstream files, which can be transferred to the FPGA accelerator to partially reconfigure one of the two applications in Figure 16(a), before requesting a computation.

2) Cluster Environment

In the future, it will be possible for each server in a cluster environment to have an FPGA accelerator, and the FPGA accelerators in some servers might sometimes be idle when others are being overused. A possible extension is to introduce a TCP/IP based

mechanism to pvFPGA. When the coprovisor recognizes that there are too many requests in the queue for the native FPGA accelerator, while the FPGA accelerators in other servers are underutilized, some of the requests can be scheduled through an extra TCP/IP mechanism to the idle FPGA accelerators in remote physical machines.

3) NUMA Architecture

Though most of today's systems use SMP (Symmetric Multiprocessing) architecture, NUMA (Non-Uniform Memory Access) architecture was introduced to surpass the scalability limit of SMP. In late 2012, Xilinx introduced QPI (QuickPath Interconnect) support for its Virtex-7 series FPGAs, which enabled them to work in a NUMA environment [64]. On the other hand, the Xen VMM already deals with NUMA in a number of ways [65]. For example, a domain is pinned to a NUMA node when it is created, and it is scheduled to run on its dedicated NUMA node to avoid remote memory access. Therefore, it is possible to port pvFPGA to NUMA architecture in the future.

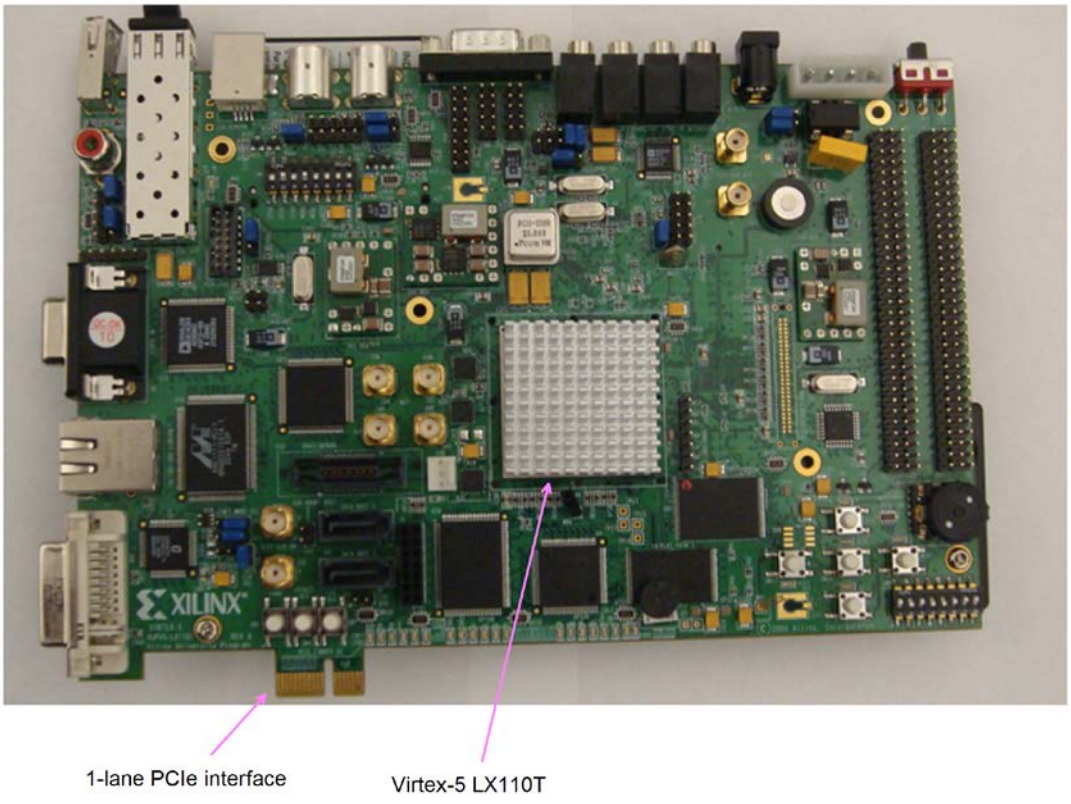
With NUMA architecture, the DMA controller can still be used for data sending and fetching. As shown in the document [64], an FPGA accelerator is positioned in a processor socket instead of a PCIe slot, so the DMA controller must use physical addresses rather than bus addresses to access the memory. Either the data pool is allocated local to the FPGA, or local to the CPU that the domain runs in, the data transfer can still be pipelined. Thus, another future direction of our work is to port pvFPGA to a NUMA machine.

Appendix A

Evaluation Platform Exhibition

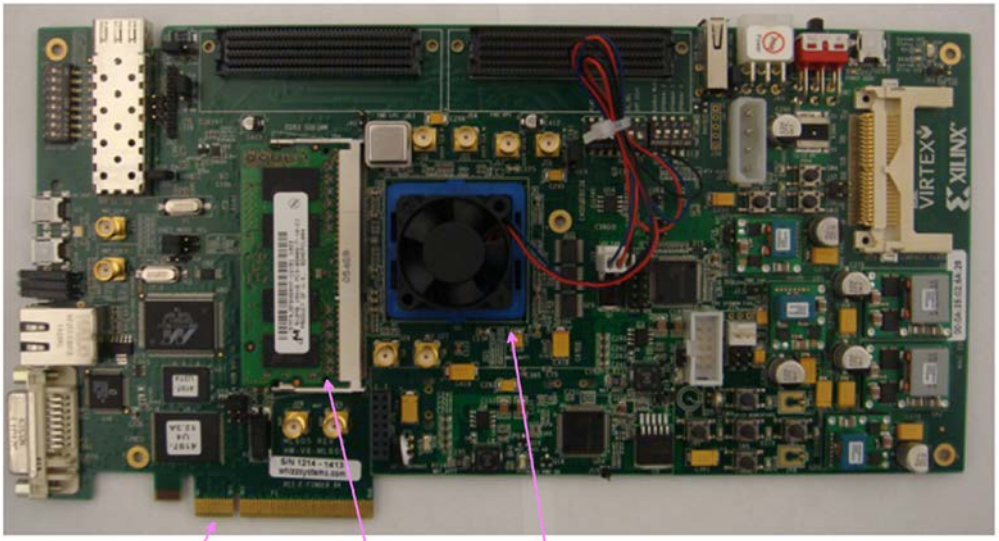
1 ML505 Evaluation Kit

1) Front Side



2) Back Side

2 ML605 Evaluation Kit



3 System Overview

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the clouds: A Berkeley view of cloud computing,” University of California, Berkeley, Tech. Rep. UCB-EECS, February 2009.

[2] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, “The Java® Virtual Machine Specification (Java SE 7 Edition)”, Oracle Corporation, February 2013.

[3] K. Yamaoka, T. Morimoto, H. Adachi, T. Koide, H. J. Mattausch, “Image segmentation and pattern matching based FPGA/ASIC implementation architecture of real-time object tracking,” Proceedings of the 2006 Asia and South Pacific Design Automation Conference, pp. 176-181, 2006.

[4] J. Taibo, V. M. Gulias, P. Montero, S. Rivas, “GPU-based fast motion estimation for on-the-fly encoding of computer-generated video streams,” Proceedings of the 21st international workshop on Network and operating systems support for digital audio and video, pp. 75-80, 2011.

[5] P. S. Lee , C. S. Lee, J. H. Lee, “Development of FPGA-based digital signal processing system for radiation spectroscopy,” Radiation Measurements, vol. 48, pp. 12-17, January 2013.

[6] X. Tian, K. Benkrid, “High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU,” ACM Transactions on Reconfigurable Technology and Systems, vol. 3, 4, pp. 1-22, November 2010.

[7] Y. Dou, Y. Lei, G. Wu, S. Guo, J. Zhou, L. Shen, “FPGA accelerating double/quad-double high precision floating-point applications for ExaScale computing,” Proceedings of the 24th ACM International Conference on Supercomputing, pp. 325-336, 2010.

[8] S. Che, J. Li, J. Lach, and K. Skadron, “Accelerating compute intensive applications with GPUs and FPGAs,” In Proceedings of the 6th IEEE Symposium on Application Specific Processors, June 2008.

[9] M. Dowty and J. Sugerman, “GPU Virtualization on VMware’s Hosted I/O Architecture,” SIGOPS Operating Systems Review, vol. 43, pp. 73-82, July 2009.

[10] G. Giunta, R. Montella, G. Agrillo and G. Coviello, “A GPGPU Transparent

Virtualization Component for High Performance Computing Clouds," In Proc. of Euro-Par, Heidelberg, 2010.

[11] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-Accelerated Virtual Machines," Proc. ACM Workshop System-Level Virtualization for High Performance Computing (HPCVirt '09), pp. 17-24, 2009.

[12] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-independent graphics acceleration," In Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, pp. 33 – 43, June 2007.

[13] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," In Proc. of the Intl. Symposium on High Perf. Distributed Computing, pp. 217-228, 2011.

[14] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU Accelerated High Performance Computing in Virtual Machines," IEEE Transactions on Computers, vol. 61, no. 6, pp. 804-816, June 2012.

[15] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?" In Proceedings of the 2005 IEEE International Conference on Field-ProgrammableTechnology, pp. 111-118, 2005.

[16] E. El-Araby, I. Gonzalez, T. El-Ghazawi, "Exploiting partial runtime reconfiguration for High-Performance Reconfigurable Computing," ACM Transactions on Reconfigurable Technology and Systems, vol. 1, no. 4, pp.1-23, 2009.

[17] E. El-Araby, I. Gonzalez, T. El-Ghazawi, "Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems," 2nd HPRCTA, pp. 1-8, 2008.

[18] I. Gonzalez , S. Lopez-Buedo, G. Sutter, D. S. Roman, F. J. G. Arribas, J. Aracil, "Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture," Journal of Systems Architecture, vol. 58, pp. 247-256, March 2012.

[19] C. H. Huang, P. A. Hsiung, J. S. Shen, "Model-based platform-specific co-design methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption," Journal of Systems Architecture, vol. 56, pp. 545-560, August 2010.

[20] E. Lübbers, "Multithreaded Programming and Execution Models for Reconfigurable Hardware," PhD thesis, Computer Science Department, University of Paderborn,

2010.

[21] M. Sabeghi and K. Bertels, “Toward a runtime system for reconfigurable computers: A virtualization approach,” Design, Automation and Test in Europe (DATE09), April 2009.

[22] C. Maxfield, “FPGAs: Instant Access”, Newnes, 2011.

[23] R. Wain et al, “An overview of FPGAs and FPGA programming; Initial experiences at Daresbury,” Technical report, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, UK, 2006.

[24] Virtex-6 FPGA Configurable Logic Block User Guide, Xilinx Corporation, February 3, 2012.

[25] R. Lipsett, E. Marschner, and M. Shahdad. VHDL – The Language. IEEE Design & Test of Computers, pp. 28–41, April 1986.

[26] Open Verilog International (OVI). Verilog Hardware Description Language Reference Manual, Version 1.0, 1996.

[27] S. Palnitkar,“Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition”, Prentice Hall PTR, February 2003.

[28] ModelSim® Tutorial Software Version 10.0c, Mentor Graphics Corporation, 2011.

[29] Partial Reconfiguration Flow Presentation Manual, Xilinx Corporation, 2012.

[30] Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes, Intel Corporation, August, 2012.

[31] C. Waldspurger , M. Rosenblum, “I/O virtualization,” Communications of the ACM, vol. 55, no. 1, pp. 66-73, January 2012.

[32] A. Vasudevan, J. M. McCune, N. Qu, L.v Doorn, and A. Perrig, “Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture,” Proceedings of the 3rd international conference on Trust and trustworthy computing, pp. 141-165, 2010.

[33] G. J. Popek,R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” Communications of the ACM, vol. 17, no. 7, pp. 412-421, July 1974.

[34] D. Chisnall, “The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series),” Prentice Hall PTR, Upper Saddle River, USA, 2007.

[35] J. Nakajima, Q. Lin, S. Yang, M. Zhu, S. Gao, P. Yu, Y. Dong, Z. Qi, K. Chen, H. Guan, M. Xia, "Optimizing virtual machines using hybrid virtualization," Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 573-578, 2011.

[36] R. P. Goldberg, "Architectural principles for virtual computer systems," Ph.D. Thesis. Division of Engineering and Applied Physics, Harvard University Cambridge Massachusetts, 1973.

[37] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," In Proceedings of the Linux Symposium, pp. 225–230, 2007.

[38] Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator," Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 41-46, 2005.

[39] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp.164-177, 2003.

[40] Xen PCI Passthrough, http://wiki.xen.org/wiki/Xen_PCI_Passthrough. Last access date: May 31st, 2013.

[41] D. E. Williams, J. Garcia, "Virtualization with Xen including XenEnterprise, XenServer and XenExpress," Syngress, May 2007.

[42] S. Senthilvelan and M. Senthilvelan,"Study of Content-Based Sharing on the Xen Virtual Machine Monitor," http://pages.cs.wisc.edu/~remzi/Classes/736/Spring2005/Projects/Muru-Selva/. Last access date: May 31st, 2013.

[43] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," In Proc. of Virtual Execution Environments, 2008.

[44] A. Keller, " kernel space - user space interfaces, " http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html#s8. Last access date: May 31st, 2013.

[45] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Drivers (Third Edition)," February 2005.

[46] R. Love, "Linux Kernel Development (Third Edition)," June 2010.

[47] J. Wiltgen and J. Ayer, "Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions," Xilinx Corporation, September 29, 2011.

[48] H. Kavianipour S. Muschter and C. Bohm, “High Performance FPGA-based DMA Interface for PCIe,” The 18th IEEE Real Time Conference (RT), pp. 1-3, June 2012.

[49] Virtex-6 FPGA Integrated Block for PCI Express, Xilinx Corporation, September 2010.

[50] Virtex-6 FPGA Memory Interface Solutions, Xilinx Corporation. June 2011.

[51] LogiCORE IP FIFO Generator v8.2, Xilinx Corporation, June 22, 2011.

[52] DMA Back-End Core User Guide, Northwest Logic Corporation, 2010.

[53] W. Huang, M. Koop, Q. Gao, and D. K. Panda, “Virtual machine aware communication libraries for high performance computing,” In Proc. of SuperComputing (SC’07), Reno, NV, November 2007.

[54] K. Kim, C. Kim, S.-I. Jung, H. Shin and J.-S. Kim, “Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen,” In Proc. of Virtual Execution Environments, 2008.

[55] J. Wang, K.-L. Wright, and K. Gopalan, “XenLoop: A transparent high performance inter-VM network loopback,” In 17th HPDC, pp. 109–118, June 2008.

[56] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin, “XenSocket: A high-throughput interdomain transport for VMs,” In Proceedings of Middleware, 2007.

[57] PCI Express Base Specification 1.1,PCI-SIG, 2003.

[58] PCI Express Base Specification 2.1,PCI-SIG, March 4, 2009.

[59] LogiCORE IP Fast Fourier Transform v7.1.Xilinx Corporation, March 2011.

[60] C. Young, J. A. Bank, R. O. Dror, J. P. Grossman, J. K. Salmon, and D. E. Shaw, “A 32×32×32, spatially distributed 3D FFT in four microseconds on Anton,” In Proc. ACM/IEEE Conf. on Supercomputing, 2009.

[61] M. Frigo and S. G. Johnson, “The design and implementation of fftw3,” Proceedings of the IEEE, vol. 93, no. 2, pp. 216–231. February 2005.

[62] H. S. Shin, K. H. Kim, C.Y. Kim, S. I. Jung, “The new approach for inter-communication between guest domains on Virtual Machine Monitor,” 22nd International Symposium on Computer and Information Sciences, pp. 1-6, November 7-9, 2007.

[63] Netfilter. http://www.netfilter.org/. Last access date: May 31st, 2013.

[64] QuickPath Interconnect IP: Enhancing Intel Processors and accelerating critical server data paths, Xilinx, 2012.

[65] Xen NUMA Introduction, http://wiki.xen.org/wiki/Xen_NUMA_Introduction . Last access date: May 31st, 2013.

[66] M. Bolic, W. Wang, J. Parri, "pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment", International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Montreal, October 2013.