NANYANG TECHNOLOGICAL UNIVERSITY

NANYANG TECHNOLOGICAL UNIVERSITY

SYSTEM-LEVEL DRIVERS FOR SOFTWARE-HARDWARE COMMUNICATION ON THE XILINX ZYNQ

by

GOVINDARAJAN MANIKANDAN
(G1402198B)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2015

Contents

List of Figures

List of Tables

Abbreviations

ACP Accelerator Coherency Port

ADC Analog-to-Digital Converter

AFI AXI FIFO Interface

API Application Processing Interface

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block random access memory

DMA Direct Memory Access

FPGA Field Programmable Gate Array

GP General purpose

GPU Graphic Processing Unit

HP High Performance

HPRC High-performance Reconfigurable Computing

LUT lookup table

OCM On-Chip Memory

OS Operating system

PL Programmable Logic

PS Processing system

PSoC Programmable System on-chip

RFS Root File System

Abstract

In the quest for hardware acceleration of applications, embedded reconfigurable platforms have emerged with significant potential for addressing the demand for performance at low power consumption. These platforms, such as Xilinx Zynq, couple one or more general purpose processors with reconfigurable fabric, where the reconfigurable fabric is used to accelerate some compute intensive tasks of an application. The major concern in such a system is the integration of the accelerator with the processor and the efficiency of the software-hardware (SW-HW) communication. Not only the integration of accelerators with an embedded processor(s) but a system-level driver is also crucial to enable communication abstraction while performing data transactions between processor and reconfigurable fabric. This report presents a number of experiments to characterize software-hardware communication interfaces on the Xilinx Zynq platform within a general purpose operating system (Linux) framework to study the effect of interface choice on the maximum performance of these interfaces. The goal is to quantify how and when these interfaces differ in terms of performance. We present an approach for sending data between DDR to DDR memory locations and DDR to BRAM locations in an optimized way achieving bandwidth close to theoretical maximum. Use of optimized DMA engine driver calls and moving the dma intialization overhead out of the write and read functionalities provided a bigger improvement in achieving a good bandwidth. This work consists of development of platform and character driver in case of PS-DMA and character driver in case of PL-DMA. Necessary hardware architecture using a memory subsystem is also created in order to establish a proper communication between PS and PL. This work concentrated on developing driver that meets the standard rules of driver development and also optimized driver in which new technique such as Zero-copy is implemented. We also measured the performance of developed PS-DMA drivers, PL-DMA drivers and provided a comparison against commercially available system-level driver, Xillybus.

Acknowledgment

I would like to express my deepest appreciation to all those who provided me the possibility to complete this report. A special gratitude I give to Assoc Prof Dr Douglas Leslie Maskell whose contribution towards suggestions and encouragement, helped me in finishing up the thesis work.

Furthermore, I would also like to acknowledge with much appreciation the crucial role of mentor Abhishek Kumar Jain, who gave me continuous support and guidance in the entire phase of my dissertation. A special thanks goes to Jeremiah Chua for lending his technical support and facilities in Center for High Performance Embedded Systems (CHiPES).

I have to appreciate the timely guidance of Vipin Kizheppatt and Shankar Shreejith and their constructive suggestions. Finally, I have no valuable words to express my thanks and gratitude to my parents and their support during my higher studies.

Chapter 1

Introduction

1.1 Motivation

In the quest for hardware acceleration of applications, embedded reconfigurable platforms have emerged with significant potential for addressing the demand for performance at low power consumption. These platforms couple one or more general purpose processors with reconfigurable fabric, where the reconfigurable fabric is used to accelerate some compute intensive tasks of an application. The major concern in such a system is the integration of the accelerator with the processor and the efficiency of the software-hardware (SW-HW) communication. Not only the integration of accelerators with an embedded processor(s) but a system-level driver is also crucial to enable communication abstraction while performing data transactions between processor and reconfigurable fabric. Managing SW-HW communication is normally one of the embedded processor's many tasks, and hence this must be done in a way that does not degrade overall system performance. Low latency and high bandwidth are the key requirements to enhance the efficiency of the hybrid system.To address possible communication bottlenecks, particularly in providing high bandwidth transfers to the reconfigurable fabric, it has been proposed to tightly integrate processors and reconfigurable fabric on a single platform. A number of tightly coupled architectures have resulted [4], including vendor specific platforms with integrated processors such as the Xilinx Zynq [2].

Reconfigurable accelerators can be integrated with processors over many different types of interface, such as PCIe, Ethernet and Advanced eXtensible Interface (AXI), etc. While the Zynq platform provides high speed AXI interfaces for communication, designers must develop a system-level driver, including a memory subsystem around the interfaces and software drivers, to manage the transportation of data to and from the accelerator via the AXI interfaces. In our work, the goal is to present a number of experiments to characterize software-hardware communication interfaces on the Xilinx Zynq platform within a Linux based framework to study the effect of interface choice on the maximum performance of these interfaces. We also aim to compare the performance of proposed drivers with a commercial system-level driver, Xillybus.

1.2 Contribution

This report presents a number of experiments to characterize software-hardware communication interfaces on the Xilinx Zynq platform within a general purpose operating system framework to study the effect of interface choice on the maximum performance of these interfaces. The goal is to quantify how and when these interfaces differ in terms of performance. On the FPGA fabric, we develop an AXI-compliant, lightweight memory sub-system (a portable bridge between the accelerators and the external memory) and on the processor, we develop Linux based drivers, specifically Direct Memory Access (DMA) drivers, to provide communication APIs. We first develop PS-DMA driver (including a platform driver) and measure the performance of driver for DDR-DDR communication and DDR-PL communication via General purpose (GP) AXI interfaces. We then develop PL-DMA drivers and measure the performance of drivers for DDR-DDR communication and DDR-PL communication via High Performance (HP) and Accelerator Coherency Port (ACP) based AXI interfaces. Finally, we compare the performance of proposed drivers with a commercial system-level driver, Xillybus.

The main contributions can be summarized as follows:

- PS-DMA Driver implementation
- PL-DMA Driver implementation

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on embedded platforms, software-hardware communication and system-level drivers. Chapter 3 studies current state of the art drivers and techniques for communication abstraction. In chapter 4, we present system architecture of proposed system-level driver and DMA drivers for communication abstraction. In chapter 5, we present experiments to measure the performance of PS-DMA driver for DDR-DDR communication and DDR-PL communication via general purpose (GP) AXI interfaces. Chapter 6, we present experiments to measure the performance of drivers for DDR-DDR communication and DDR-PL communication via high performance (HP) and Accelerator coherency port (ACP) based AXI interfaces. We conclude in chapter 7 and discuss future work.

Chapter 2

Background

2.1 Embedded Reconfigurable Platforms

Today's trend in the field of Embedded reconfigurable platforms is that they couple one or many processors with reconfigurable hardware, such as Field Programmable Gate Array (FPGA). Few examples are, Zynq-7000 All programmable SoC from Xilinx [2], Micorsemi's SmartFusion SoC, Cypress's Programmable System on-chip (PSoC) etc. The major advantages of these platforms are reconfigurability, reduced power consumption than Graphic Processing Unit (GPU) and many-core processors, real-time execution capability, and most importantly, high spatial parallelism which can be used to significantly accelerate complex algorithms [5][6]. Solutions like Application Specific Integrated Circuit (ASIC) have high-performance benefits but falls back in flexibility metric. On the other hand, solutions like embedded reconfigurable platforms are highly flexible.

Having talked about the advantages and evolution of reconfigurable platforms, it is necessary to study and understand the difficulties in the implementation of application on this kind of platforms. The main challenges are, communication interface between the main processor and reconfigurable fabric, memory architecture and management, support for services such as communication, synchronization, scheduling, interrupt handling through efficient operating systems and effective management for reconfiguration of fabric and tools for mapping application to the platform.

2.1.1 Example: Xilinx Zynq

Unlike general purpose processing systems, FPGAs are in a unique position to take the maximum advantage of Moore's Law improvements in semiconductor technology [7]. The concept of reconfigurable hardware existed right from 1960ś [8] but the commercially successful attempt was made by these market leaders. Both major FPGA vendors, Xilinx and Altera, have recently introduced reconfigurable platforms consisting of high performance processors coupled with programmable logic. These platforms partition the hardware into a processor system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the programmable logic (PL) where custom hardware can be implemented. The two parts are coupled together with high throughput interconnect to maximize communication bandwidth. The principle is having the concept of main processor that controls an array of reconfigurable hardwares to perform compute-intensive tasks. The focus of this report is on Xilinx Zynq SoC [2].

Presence of both processor and reconfigurable fabric on Zynq platform seems beneficial due to several reasons. This approach is usually processor centric for easy control and adaptability. Zynq follows a processor centric approach by allowing the processors to boot first. FPGA fabric provides application specific acceleration and power of reconfigurability while the processor handles the control intensive tasks. Also FPGA based designs are more energy and power efficient, smaller and more flexible when compared to SW implementation, which makes it more suitable for embedded systems. Fig. 2.1 shows some use cases for Zynq:

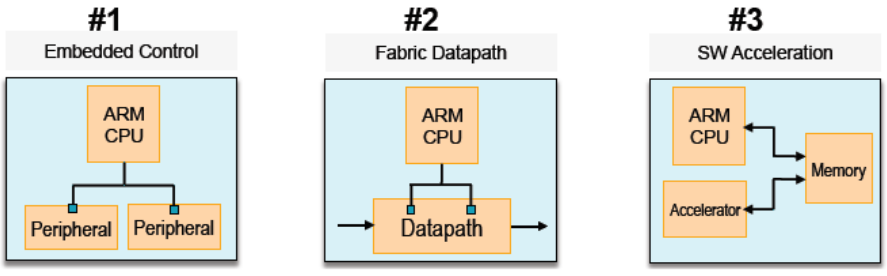

Figure 2.1: Zynq Use Cases Diagram

The Xilinx Zynq product integrates dual-core ARM Cortex A9 cores and Xilinx Programming Logic [PL] in a single device built on a state-of-the-art with on-chip memory, external memory interfaces and a rich set of I/O peripherals [2]. The Processing system (PS) contains a dual-core ARM Cortex-A9 processor which can boot independently unlike other cores such as PowerPC. The PS also consists of a double-precision floating point unit, commonly used peripherals, a dedicated hard DMA controller (PS-DMA), On-Chip Memory (OCM) and external memory interfaces. Each processor is a low-power and high-performance core that contains 32 KB Level-1 separate caches for instruction and data. The components of PS are listed as follows:

- Two ARM Cortex-A9 cores with ARMv7 ISA, with run-time options to configure as a single processor, asymmetric or symmetric multiprocessor
- NEON 128b SIMD coprocessor and VFPv3 per MPcore.
- 32KB instruction and L1 data caches per core
- 512KB shared L2 cache
- Snoop Control Unit (SCU) and the ACP for maintaining L1 and L2 cache coherency.
- 256 KB of On-Chip SRAM (OCM) with parity
- A set of IO peripherals
- DDR controller with three major blocks - AXI memory port interface, transaction scheduler, digital PHY.
- DMA controller for PS [four channels] and PL [four channels].
- Interconnect has three separate features for different purposes- AXI high performance data switches, based on ARM NIC-301, PS-PL interfaces.

The Programmable Logic (PL) consists of Artix 7 FPGA fabric. It has 6 input LUTs and 36kb Block RAMs which can be configured as two 18 kb blocks. This family is integrated with the three smallest devices belong to zynq family and they are 7Z010, 7Z015 and 7Z020. The processor in the system always boots first and PL can be configured as a part of boot process or at some point of time in the future. In addition to this, PL can be reconfigured completely or partially using partial reconfiguration

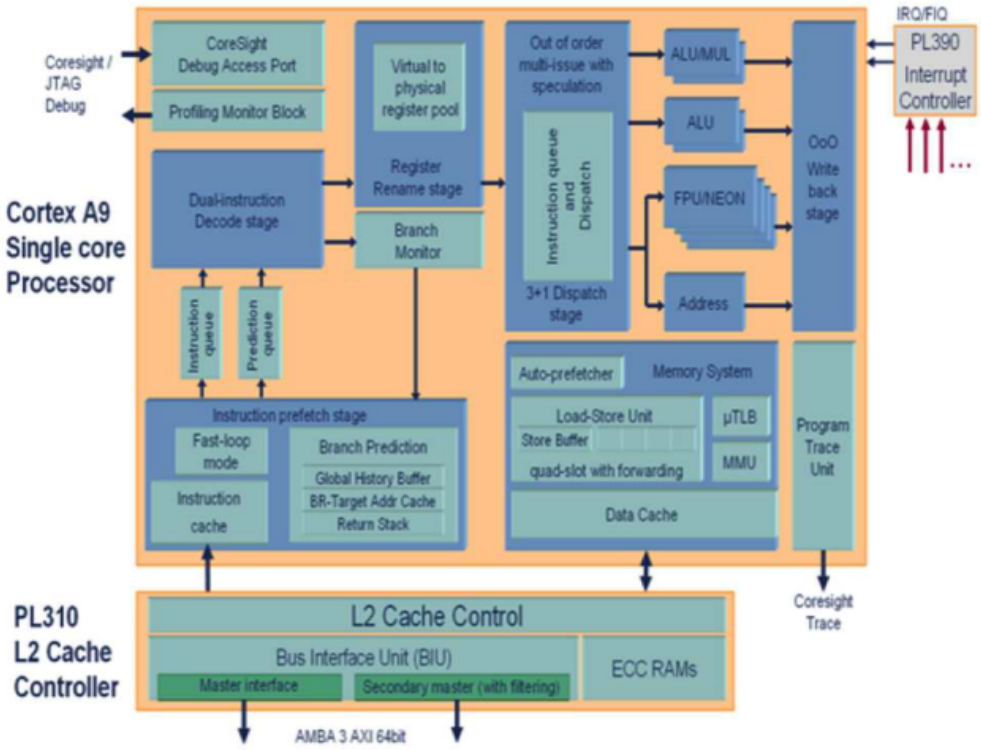

Figure 2.2: Processing System (PS) Block Diagram[1]

(PR) feature. This is analogous to the dynamic loading and unloading of software modules. The PL configuration data is referred to as bitstream. The predictable latency feature of PL is useful for real-time applications. Power management can be achieved by powering down the PL as it is on different power domain than the PS. The PL provides a rich architecture of user-configurable capabilities and they are [2]:

- Configurable logic blocks (CLB) with 6-input lookup table (LUT)
- 36KB of block RAM with dual port capability
- Digital signal processing - DSP48E1 Slice with optional pipelining, ALU and dedicated buses for cascading.
- Low skew and low jitter clock distribution
- Configurable high performance I/Os.
- Dual Analog-to-Digital Converter (ADC) blocks with 12-bit and 1 MSPS rate

2.2 Software-Hardware Communication

High-performance Reconfigurable Computing (HPRC) is getting popular due to the combination of main processors with reconfigurable computing-based accelerators like FPGAs [8]. Major concern, in both static and dynamic reconfigurable systems, is data transfer bandwidth. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the CPU and the reconfigurable fabric, it has been proposed to more tightly integrate the processor and the reconfigurable fabric. A number of tightly coupled architectures have resulted [4, 9], including vendor specific systems with integrated hard processors. These platforms have a large number of embedded memories to avoid memory bottlenecks. This created opportunities to exploit high speed interfaces and raised questions of how to organize, manage, and exploit these embedded reconfigurable memories. A data-transport mechanism using a shared and scalable memory architecture for FPGA based computing devices was proposed in [10]. It assumes that the FPGA is connected directly to L2 cache or memory interconnect via memory interfaces at the boundaries of the reconfigurable fabric. Similar mechanism of data-transport was demonstrated in a virtualized framework on the Xilinx Zynq [11, 12]. The increase in the PL logic resources makes it possible to execute compute-intensive tasks in a faster manner. Sooner, this trend makes a big impact in the embedded platforms as well. As pointed out earlier, major reason for this change is due to the processing power of PL logic. This implicitly explains the need for data transfer between processor and PL logic which should be highly optimized and should give maximum data bandwidth to have a good system performance.

When reconfigurable hardware is tightly coupled with a processor with memory management unit (MMU) support, reconfigurable hardware can share processor's MMU. The processor can now be used by the OS to perform memory accesses and then it can feed data to reconfigurable HW for computation. This model brings good control but reduces the ability of the processor to act as a compute unit as is kept busy in memory transactions. DMA controllers are normally used in such cases to counter this issue of handling memory transactions.

2.2.1 Communication Interfaces on Zynq

Xilinx-Zynq family follows the AXI protocol from ARM introduced in 1996. The interfaces between PS and PL are based on AXI protocol and they are of three types for different purposes [13],

- AXI4 - For high-performance memory mapped application
- AXI4-Lite - For simple, low-bandwidth memory mapped application
- AXI4-Stream - For high speed streaming data application

Memory mapped applications often provide a more homogeneous way to view the system because the peripherals operate in a defined memory mapped address, whereas in the case of streaming applications which focuses on data-centric and data-flow paradigm where there is no concept of address. The interconnection between PS and PL provided by the AXI bus facilitates any logic implemented in PL to be addressable by PS thereby acting as a memory-mapped peripheral. AXI [13] is an asynchronous interface with independent read and write channels. This interface provides low latency as well as processor DMA access to the peripherals. AXI interfaces have a common principle of sending data between a single AXI master and a single AXI slave and they contain five different channels - Read and write Address channels, Read and write Data channels and write Response channel. Data can move in both the directions simultaneously and data transfer size can vary with different protocols for example, AXI4 has a burst size of up to 256 whereas AXI4-Lite has only one data transfer per transaction. With respect to the underlying PL fabric, the AXI interfaces used for the following ports,

- AXI_GP - Two master and two slave interfaces
- AXI_HP - Four slave interfaces with access to DDR and OCM
- AXI_ACP - One slave interface for cache coherent memory access

GP port

AXI_GP port includes the following feature set,

- Data bus width - 32-bit

- Master port ID width - 12 with 8 read and 8 writes issuing capability.
- Similarly Slave port has port ID width of 6 and 8 read-write acceptance capability.
- Port normally connects with PS-DMA present in the processor side and the underlying PL logic.
- The maximum bandwidth achieved with GP ports when 32-bit data sent to PL logic running at 100 MHz is 400 MB/s and with increased frequency at 150 MHz is 600 MB/s

HP port

There are four HP interfaces that provide PL bus masters with high bandwidth transfers to DDR and OCM memory in the platform. Each interface consists of two separate buffers for read and write traffic and they often referred to as AXI FIFO Interface (AFI). Diagram of HP connectivity is shown in Fig. 2.3 and the main features are listed as follows,

- Data bus width - 32 or 64-bit
- Four HP ports are available that can support maximum of four masters to drive the ports .
- Write channels can be configured to store and forward write commands or allow without any storage.
- QoS priority option is available to assign an arbitration priority to the read and write commands.
- Port normally connects with Soft IP core DMA provided by Xilinx such as AXI-DMA, AXI-CDMA and AXI-VDMA.
- The maximum bandwidth achieved with a single HP port when 64-bit data transaction is done with the logic running at 100 MHz is 800 MB/s and with increased frequency at 150 MHz is 1.2 GB/s
- By using the four available ports for the data transaction the maximum bandwidth attained is 3.2 GB/s running at 100 MHz and 4.8 GB/s

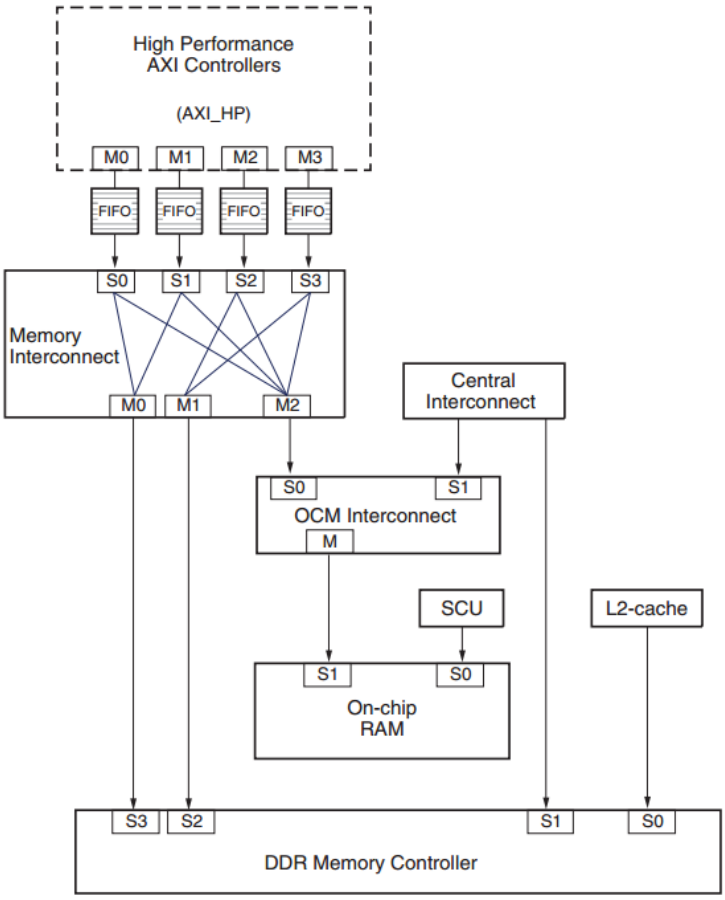

Figure 2.3: HP port Connectivity Diagram[2]

ACP port Zynq platform has only one specialized port for doing cache coherent transactions between with processor caches and PL logic. Diagram of ACP connectivity is shown in Fig. 2.4 and the major features of this port are as follows,

- Data bus width - 64-bit
- Single ACP port is available that can support low-latency transfers with optional coherency between L1, L2 caches and PL logic memory.
- From the Fig.2.4 it is clear that ACP port is connected directly to the snoop control unit of the processor which is responsible for the cache-coherency.
- With the proper configuration of this port, it is possible to achieve very high data bandwidth and also with the improper handling of ACP port will degrade

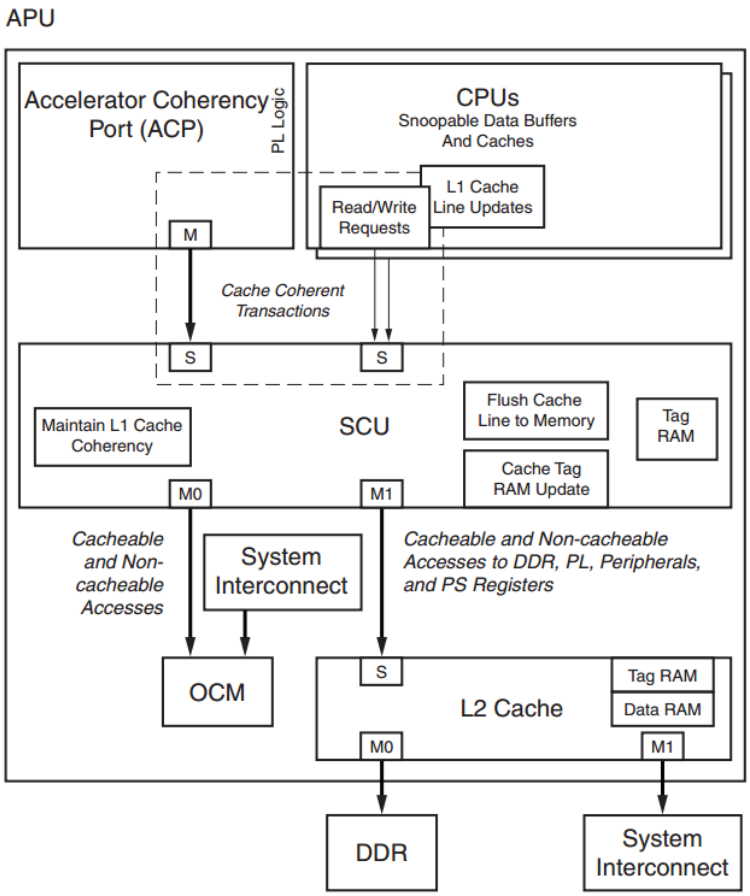

Figure 2.4: ACP port Connectivity Diagram[2]

the entire system performance.

- ACP transfer are optimized when it matches the core's coherent requests.
- Port normally connects with Soft IP core DMA provided by Xilinx such as AXI-DMA, AXI-CDMA and AXI-VDMA.
- The maximum bandwidth achieved with a single HP port when 64-bit data transaction is done with the logic running at 100 MHz is 800 MB/s and with increased frequency at 150 MHz is 1.2 GB/s

2.2.2 Communication Abstraction

Communication abstraction is a method to represent communication interfaces and memory sub-system at a logical view by providing an interface similar to SW Application Processing Interface (API). This logical view basically decouples the functionality of communication interfaces from their actual implementation. Communication abstraction might be required by system developers to ensure standards-compliance, handle the multitude of communication protocols, and reduce developer effort. Communication abstraction should be an integral part of PSoC where there is a need of easy and isolated memory transactions between processor and reconfigurable fabric.

The key figure of merit in embedded reconfigurable platforms is communication abstract at both hardware and software level. In case of hardware, the abstraction can be provided through hardware logic around the AXI port that gives maximum bandwidth to transfer the data between processor and PL logic. As the applications has to send data in order to be transferred to PL logic and for this reason the application should know the functions or API in an abstract form at the software level to send and receive data.

The key attraction of communication abstraction techniques is their capability to seamlessly access memory sub-system and abstraction of physical interfaces. An abstracted memory sub-system should be an integral part of programmable SoC where there is a need for easy and isolated memory transactions[14]. System designs can incorporate these abstractions at platform level to make use of reconfigurable memory as local memory space for application accelerators. It works the same way as cache works for processors. As both Altera and Xilinx started embedding hard memory blocks (Block random access memory (BRAM)) within fine grained reconfigurable fabric, researchers also started to explore these blocks as distributed storage elements.

System level drivers give a clear abstraction to the application developers. As a developer, there is a need to understand the functionalities of each abstracted functions, then use it and get the results without any concern about the underlying drivers and hardware. In this report, we present the proposed memory subsystem by exploring how BRAMs can be used as local and distributed memory space to provide seamless access of data to streaming accelerators. We also explore the ways to provide

OS support for the memory sub-system and communication abstraction. Different techniques have been explored in literature to develop such an abstracted memory sub-system, including Garp[4], CoRAM[10] and PyCoRAM[15].

2.3 Operating System Support

A number of researchers have focused on providing OS support for reconfigurable hardware so as to provide a simple programming model to the user and effective run-time scheduling of hardware and software tasks [16, 17, 18, 19]. A technique to abstracting reconfigurable co-processors in high performance reconfigurable computing (HPRC) systems was presented in [20]. ReconOS [21] is based on an existing embedded OS (eCos) and provides an execution environment by extending a multi-threaded programming model from software to reconfigurable hardware. Several Linux extensions have also been proposed to support reconfigurable hardware [22, 23]. RAMPSoCVM [24] provides runtime support and hardware virtualization for an SoC through APIs added to Embedded Linux to provide a standard message passing interface.

Usage of Operating system (OS) in reconfigurable platforms renders help in coordination of multiple hardware tasks and efficient management of memory, shared resources among the applications[25]. In particular the presence of open-source powerful OS such as Linux provides better control over the communication interfaces, scheduling of threads and tasks, synchronization, interrupt management etc. When compare with bare-metal applications, Linux application does not perform efficiently but they are heavily abstracted from the underlying hardware and scheduling which provides an easy way for the application developers to dwell upon Linux. Unlike bare-metal application that requires explicit handling of resources, synchronization, communication that creates an overhead for the developers to look into each one of this tasks and it poses a great challenge as well. Some of the examples for OS support are, SIRC[26], RIFFA[27][28] and Xillybus[3].

2.3.1 Xillinux

Commercially available linux distro for Zynq platforms is Xillinux[29] which is a combination of software and FPGA code kit based on Ubuntu 12.04 specifically meant for Zynq platforms. The major things needed for booting up the board with Xillinux are as follows,

- SD card with minimum 4GB of memory size
- Download the .img file from the distro website.
- copy the image using dd command present in linux which automatically creates two partitions.
- First one is using FAT32 format and is for copying all the bootable files such as .bin, .dtb, .bit files in order to boot the platform.
- Second one is using EXT4 format which is used for copying the Root File System (RFS) of the distro.

Setting up of SD card with Xillinux distro is very easy as it requires very minimal steps and the default image used in our experiments is Xillinux-1_3.img. This image has in-built support for xillybus architecture, Xilinx DMA IP cores and for generic DMA engine driver.

2.3.2 Xillybus

Xillybus is a portable and straightforward, DMA-based solution for data transport between processor and FPGA. It is designed to work with the interfaces: the PCIe interface (in a typical x86 based system) and the AXI interface (in an ARM based system), as the underlying transport mechanism. Standard FIFOs are provided as interfaces to the application logic on the FPGA. Each FIFO stream is mapped to a device file by a universal Xillybus driver. It provides all necessary HW-SW communication interfaces such as memory mapped to stream interface, memory mapped to memory mapped interface, memory mapped register interface. The main benefit of using Xillybus, as a memory abstraction, is that it works on Xillinux OS (lightweight management abstraction).

Xillybus is intended to do data transaction between ARM and PL logic with minimum latency and maximum data bandwidth. As stated earlier this is provided along with Xillinux distro and this can be noticed by doing $ls - lt/dev/xillybus*$ which prints out the pipes created for the xillybus usage. Since the xillybus drivers are staged and enabled by default and so it is created as in-built module in the kernel. The underlying hardware uses Xillybus hardware IP core connected through ACP port for maximum transfer between DDR memory and PL logic memory.

In the case of application users the pipe present under */dev* directory should be used according to the functionality that it provides. For example, xillybus_write_32 pipe provides an option for the application to send 32-bit data samples to the xillybus bridge. Similarly, xillybus_read_32 pipe provides facility to read 32-bit samples from the xillybus.

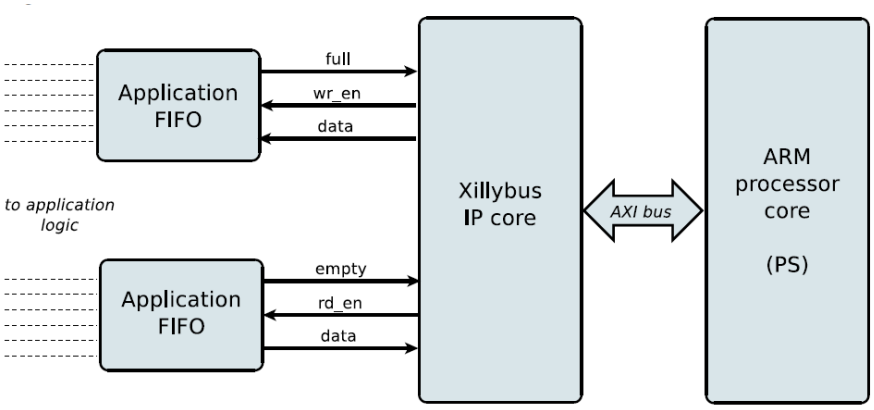

Figure 2.5: Xillybus Block Diagram[3]

Fig. 2.5 shows the abstracted view of the xillybus hardware structure. FPGA demo bundle used for Zynq consists of Integrated Software Environment and Xilinx Platform Studio project, boot.bin and device-tree files which can be modified to produce different hardware and bitstream according to the application requirements.

Chapter 3

Literature Survey

A memory sub-system is crucial to decouple the functionality of communication interfaces from their actual implementation. Instead of developing an application-specific memory sub-system around an accelerator, CoRAM was proposed as a communication abstraction mechanism using a shared and scalable memory architecture [10]. It provides an interface to off-chip memory using on-chip interconnect generated by the CONNECT NOC generator [30]. Communication between off-chip memory and on-chip memory is abstracted and can be controlled using software threads. PyCoRAM [15] is a python based automation infrastructure of the CoRAM project with slight modifications. For example, instead of making use of CONNECT NOC generator, it uses AMBA AXI-4 infrastructure and provides support for AXI based IP and hence it is suitable for platforms like Zynq. LEAP[31] (logic-based environment for application programming) scratchpad was proposed as an automatic memory management system to make reconfigurable memory hierarchy invisible which is very similar to the concept of CoRAM. Both of these projects share the objective of providing a standard memory abstraction by virtualizing an FPGAs memory and I/O interfaces. LEAP abstracts away the details of memory management by exporting a set of interfaces to local client address spaces.

SIRC [26] was proposed as an open source OS based abstract interface (a software API and hardware interface) for communication between a PC and the FPGA. SIRC only supports Windows x86 based platforms and provides a peak bandwidth of 118

MB/s. RIFFA [28] was proposed as an open source reusable framework to integrate the FPGA IP cores to workstation software using the PCIe interface. Currently, RIFFA only supports the PCIe interface hence it is not possible to use it on the AXI based Zynq platform. A more comprehensive survey of various communication frameworks is given in [32] with a focus on FPGAs in general purpose computers. DyRACT [33] was recently proposed to provide SW APIs and Linux drivers for managing partial reconfiguration and high throughput data communication over PCIe interface. More detailed study about the usage of AXI-DMA and different ways of configuring using register mode and scatter gather mode is done in the [34]. Their main idea is to use AXI-DMA as a memory manager for supporting irregular sparse access patterns and to show it is faster than traditional cache based access on embedded CPUs on the Xilinx Zedboard.

HybridOS [35] was developed as a set of Linux extensions to study the methods of data communication between tasks running on processor and accelerators. Authors presented four accelerator access methods out of which two are DMA based and another two are non-DMA based. Authors have performed a case study in which JPEG application was implemented on FPGA fabric and data transfer was done using proposed four different methods. They have shown that most effective method of data transfer depends on data transaction size and the number of times the application will use an accelerator.

This paper [36] deals with the methods that can be used for transferring data between DDR and accelerator memory and it concentrates mainly on HP and ACP ports. It also deals with the study of memory transfer when the target, here Xilinx Zynq APSoC, is loaded with dummy tasks and also at idle. Results they could able to achieve with HP port is around 700 MB/s and 707 MB/s with ACP port. This paper does not deal with the usage of multiple HP ports and they mainly concentrated on drawing conclusions about the performance given by these ports at specific applications and when to use.In this paper, we focus on high throughput data communication over different AXI interfaces of Zynq platform without considering overheads incurred by Linux OS abstractions.

Chapter 4

System-Level Drivers for Communication Abstraction

On the FPGA fabric side, there should be local memory space within programmable logic region which can be accessed by both processor and the programmable logic region. System designs can incorporate communication abstractions at platform level to make use of reconfigurable memory as local memory space for application accelerators. It works the same way as cache works for processors. These solutions are getting popular in reconfigurable community because memory access is a big issue for programmable logic. As both Altera and Xilinx started embedding hard memory blocks (BRAMs) within fine grained reconfigurable fabric, researchers also started to explore these blocks as distributed storage elements. On the processor side, there should be a set of APIs containing system calls (in the form of software drivers) for communication between the processor and programmable logic.

4.1 Hardware Infrastructure

The Xilinx Zynq-7000, ARM based reconfigurable system, uses multiple AXI interfaces for SW-HW communication between the processor system (PS) and the programmable logic (PL). Each interface consists of multiple AXI channels, enabling

high throughput data transfer between the PS and the PL, thereby eliminating common performance bottlenecks for control, data, I/O, and memory. The available AXI interfaces to the fabric include:

AXI_GP – Two 32-bit master and two 32-bit slave AXI GP interfaces

AXI_HP – Four 64-bit/32-bit configurable, buffered AXI HP slave interfaces with direct access to DDR and on chip memory

AXI_ACP – One 64-bit AXI accelerator coherency port (ACP) slave interface for coherent memory access

We have used the Xilinx Embedded Development Kit (EDK) for our system design. Xilinx Platform studio (XPS) can be used to automatically generate custom AXI based peripherals. A peripheral connects to the AXI interconnect through the corresponding AXI IP interface (IPIF) modules, which provides a quick way to implement an interface between AXI interconnect and the user logic. A peripheral can have either a slave interface or a master interface. A slave interface is typically required by most peripherals for operations like logic control, status report, etc. A master interface is typically required by complex peripherals like DMA. XPS also provides a bus functional model (BFM) simulation platform so that the designer can verify the functionality of the generated peripheral. We generate the following custom peripherals in the PL using XPS base system builder (BSB) in order to communicate with the PS:

AXI-lite based Register peripheral - A user specific SW accessible register (up to 32 registers) interface for operations such as logic control, status, etc.

AXI based Memory peripheral - User specific memory regions (up to 8 regions) providing local storage of data in the PL. The peripheral supports burst transfer by default. This feature provides for higher data transfer rates when using the DMA controller for transactions. Fig. 4.1(a) shows one example of the memory peripheral having 4 dual port block RAMs.

Peripherals mentioned in the previous section can be deployed under various settings in the system. Fig. 4.1(b) shows a use case scenario of the proposed memory sub-system. The slave interface of the register peripheral is connected to the GP master interface via AXI interconnect and master interface of PL-DMA is used to

connect the slave interface of memory peripheral with the ACP slave interface. The control and status interface for the virtualized accelerator is provided via the register peripheral and multiple streaming I/O interfaces are provided via memory peripheral. In the memory peripheral, one port of the dual port block RAMs is connected to AXI IP interface (IPIF) while the other port is used to create streaming interfaces for the virtualized accelerator. The memory peripheral streaming I/O interfaces can be connected to the accelerator via a customized interconnect or can be connected in a fixed manner. In the next section, we characterize of the various mechanisms of interfacing the register and memory peripherals to the system via different AXI interfaces.

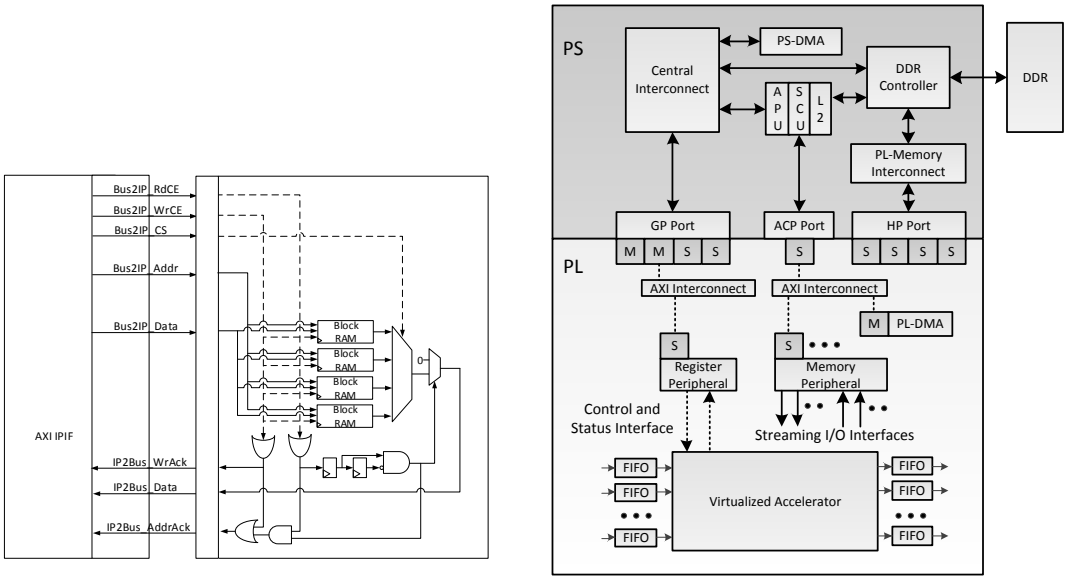

Figure 4.1: (a) AXI4 based Memory Peripheral, and (b) interfacing with the system via AXI interfaces.

4.2 DMA Drivers

The current architecture is described in the Fig. 4.2 where there are three levels of driver currently available.

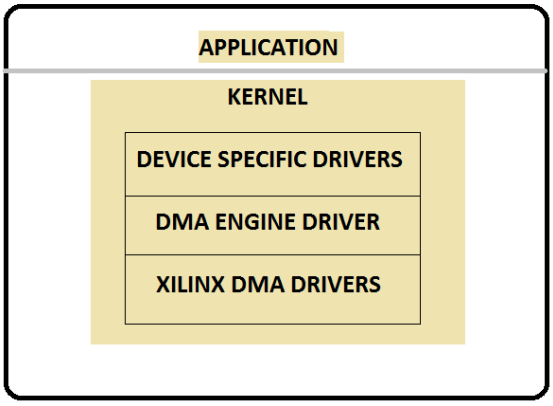

Figure 4.2: Linux overall architecture.

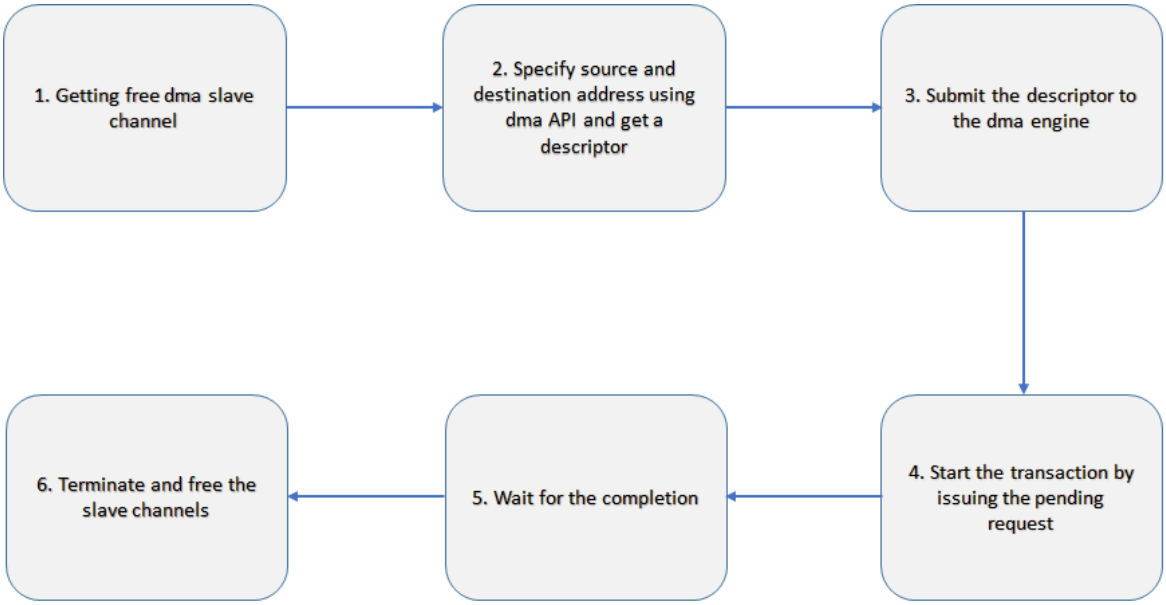

Figure 4.3: DMA driver steps

- Device Specific Drivers - contains driver that uses the dmaengine function calls for the purpose of data transaction.
- DMA Engine Driver - generic dma-engine present in the Linux kernel.
- Xilinx DMA Drivers - drivers for Xilinx dma, cdma and vdma and it is enabled by default in the Xillinux kernel.

Our major driver code belong to the device specific drivers wherein we will write either a character driver or a combination of platform and character driver [37] in order to perform the transactions. The general procedure for writing this driver is shown in the Fig.4.3.

According to the different types of driver different DMA APIs will be used but the procedure remains same [38].

4.2.1 PS-DMA Driver

With regard to PS-DMA driver, we will be writing a combination of platform and character driver along with the generic dma engine to perform transactions. With regard to DDR-DDR communication, there are two ways of performing the transactions.

The first way of doing transaction is described in the following steps [39],

- First step is to get the free dma slave channel using the API, dma_request_channel () by setting the mask alone to slave characteristics which will make sure to get the free channel from the main dma controller.
- Second step is to use the dma_async_memcpy_buf_to_buf () API to start copying the data from source to destination memory address and the return value of this is a dma cookie.
- Third step is to wait for the transfer operation to complete using the API dma_async_is_tx_complete () by passing the obtained cookie value in a while loop.
- Final step is to terminate all the dma operations using dmaengine_terminate_all () API and release the obtained dma channel using dma_release_channel () API.

Second way of performing the transactions follows the same steps (First and final step) however second and third step got replaced.

In second step, the dma_device− >prep_dma_memcpy () API is used instead of dma_async_memcpy_buf_to_buf () API.

Third step is replaced by the initialization of callback and checking for the interrupt.

There is an additional step to submit the descriptor using dmaengine_submit () API and issue pending signal to dma using dma_async_issue_pending () API which actually starts the operation and wait for its completion by checking the callback function. dma_async_memcpy_buf_to_buf () API uses cookies to check the status of transfer and dma_device− >prep_dma_memcpy () API uses interrupts for checking the acknowledgement from dma for the completion of the operation. We expect the later to be better.

When the communication is between DDR and peripheral memory locations such as PL-BRAM then getting the DMA to work will follow the below sequence (the basic sequence which can be improved by few optimizations, mentioned later in the section 5)

- First step is to get the free dma slave channel using the API, dma_request_channel () by setting the mask alone to slave characteristics which will make sure to get the free channel from the main dma controller.
- Second step is to allocate coherent memory in dma context in order to do write operation from DDR to BRAM using dma_alloc_coherent () API.
- Third step is to use the dma_device− >prep_dma_memcpy () API to inform DMA about the source and destination memory address and the return of this API give us a descriptor which will be used to start the dma operation. It is not possible to use dma_async_memcpy_buf_to_buf () API for DDR-PL communication.
- Fourth step is to initialize callback in order to get the acknowledgment from dma for completion of its operation.
- Fifth step is to submit the descriptor using dmaengine_submit () API and issue pending signal to dma using dma_async_issue_pending () API which actually starts the operation and wait for its completion by checking the callback function.
- Final step is to terminate all the dma operations using dmaengine_terminate_all () API and release the obtained dma channel using dma_release_channel () API.

DMA Driver with MMAP - in which a portion of actual physical memory is being

mapped directly from the user-space using the mmap system call and a corresponding virtual address is given to the application to write the samples [40]. This provides a way of removing the latency/overhead of copying the data from user space to kernel space. This is part of optimization which heavily improves the performance. The same procedure is followed for PS-DMA driver in the case of DDR-DDR transactions but for DDR to peripheral memory the second step is removed as the copy of user space data to the kernel is avoided and the rest of the steps are unchanged. This mmap approach is commonly referred to as Zero-copy[41] design and it is extensively used in many applications such as dma transfer, video buffers, and network packet buffers.

The main idea behind this approach in our case is we have DDR memory available as a file under $/dev/mem$. This is the major advantage for the application developers to make use of this pipe to get memory mapped address and writing to this address is equivalent to writing it to the actual physical DDR address [42]. The flip side of this approach is the physical address has to be known by the user and it is not portable since with different versions of Zedboard it is quite natural to have different address space for DDR but for the maximum performance this is the best approach it is available. So with every new version of Zedboard this address has to be validated before the usage. Another flip side is the address whichever is being memory mapped in the application has to be matched with the device driver addresses. This means if the application is using a physical address let say $0x100A0000$ for writing the samples and the same address should be accessed in the device driver too which makes the transactions to be correct.

4.2.2 PL-DMA Driver

This portion deals with the driver using soft DMA implemented in PL logic.

DMA Driver using CDMA - Central DMA [CDMA] is a soft IP core present in the PL side which can be used for transferring data from DDR-DDR transactions as well as DDR-BRAM transactions [43]. The advantage of this IP core is that it connects the peripheral with memory interconnect through high performance [HP]

ports which gives a good performance. It supports both 32-bit and 64-bit transfers with maximum burst size of 256 and also the PL logic can be made to run at 100 or $150MHz$. The maximum bandwidth that can be achieved with 32-bit running at 100 MHZ is 400 MB/s and with 64-bit is 800 MB/s. It is also possible to use all the four HP ports with CDMA transfers running at 100 MHZ with 64-bit data samples to achieve about 3.2 GB/s theoretically. Xilinx already provided the platform drivers for this IP core and it has been integrated with the Xillinux kernel. We have two options whether to have it as loadable kernel module or as an in-built module. The procedure for using the CDMA driver is same for both DDR-DDR and DDR-PL transactions and it has the following steps,

- First step is to get the free dma slave channel using the API, dma_request_channel (). In this case, the mask is specified with slave characteristics along with direction as DMA_MEM_TO_MEM and the match attribute in order to get the channels from CDMA and not from the main dma controller.
- Second step is to use the dma_device− >prep_dma_memcpy () API to inform DMA about the source and destination DDR memory address and the return of this API give us a descriptor which will be used to start the dma operation. In this case, no need to initialize callback since the completion of the DMA transactions will not depend on interrupts.
- Third step is to submit the descriptor using dmaengine_submit () API and issue pending signal to dma using dma_async_issue_pending () API which actually starts the operation.
- Fourth step is to wait for the completion of CDMA as the interrupt and its callback is not used. Each CDMA IP core has its own status register present at [BASE_ADDRESS+04h] and the value is being polled for a specific bit which corresponds to the idle or busy state in order to make sure the transaction is completed or not.
- Final step is to terminate all the dma operations using dmaengine_terminate_all () API and release the obtained dma channel using dma_release_channel () API.

Similar steps are being followed for the DDR to peripheral memory transactions

except for the destination or source address has to be changed according to the necessity of the application. The applications implemented using this driver, are all memory-mapped and the maximum performance is achieved with that.

DMA Driver using AXI-DMA - This is another soft IP core provided by Xilinx specialized for streaming data transactions between DDR storage locations via PL [44]. The main principle in transaction is that it converts the memory mapped data from DDR to PL in a streaming fashion through memory map to stream interface and also vice-versa it has stream to memory map interface. Similar to CDMA it supports 32 and 64-bit transfer with the maximum burst size of 256. Xilinx already provided the platform driver for this IP core and it has been integrated with the Xillinux kernel. We have two options whether to have it as loadable kernel module or as an in-built module. In this work, we use the platform driver as loadable module and we load whenever the experiments are carried out. The maximum bandwidth that can be achieved with 32-bit running at 100 MHZ is 400 MB/s and with 64-bit is 800 MB/s. It is also possible to use all the four HP ports with AXI-DMA transfers running at 100 MHZ with 64-bit data samples to achieve about 3.2 GB/s theoretically. But the current version of driver code available in the kernel is not supporting multiple port capability [45]. Also in our experiments we made a loopback hardware where the data whatever is fed is taken back through AXI-DMA interfaces. The procedure to use this driver in our character driver is as follows,

- First step is to get the free dma slave channel using the API, dma_request_channel (). In this case, we need two channels and the mask is specified with slave characteristics along with direction as DMA_MEM_TO_DEV and DMA_DEV_TO_MEM and the match attribute in order to get the channels from AXI-DMA and not from the main dma controller.
- Second step is to use the dma_map_single API to map DDR address one for source and another for destination in order to tell the DMA to transact the data with these addresses. This procedure is quite different as these API belong to streaming DMA operations [46].
- Third step is to use the dmaengine_prep_slave_single () API to inform DMA about the source DDR memory address with the direction as DMA_MEM_TO_DEV

and the return of this API give us a descriptor which will be used to start the dma operation. This belong to the transmission of data from DDR to PL. In the next statement use the same API to receive the data from AXI-DMA to DDR but with different DDR address and DMA_DEV_TO_MEM direction. In this case, no need to initialize callback since the completion of the DMA transactions will not depend on interrupts.

- Fourth step is to submit the descriptor using dmaengine_submit () API and issue pending signal to dma using dma_async_issue_pending () API which actually starts the operation.
- Fifth step is to wait for the completion of AXI-DMA as the interrupt and its callback is not used. Each AXI-DMA IP core has its own status register present at [BASE_ADDRESS+04h] for memory-mapped-to-stream and at [BASE_ADDRESS+34h] for stream-to-memory-mapped operation and the value is being polled for a specific bit which corresponds to the idle or busy state in order to make sure the transaction is completed or not.
- Final step is to terminate all the dma operations using dmaengine_terminate_all () API and release the obtained dma channel using dma_release_channel () API.

Chapter 5

PS-DMA based Communication

5.1 Introduction

Our platform is running with Linux OS-Xillinux kernel version3.12 from Xillybus in which the PS-DMA support is not enabled by default. This is identified by checking /proc/config.gz file on the running zed board. After configuring the support for PS-DMA and kernel recompilation, we got the DMA up and running as we can observe in the kernel boot log,

```
1 root@localhost: dmesg | grep pl330
2 [0.602915] dma-pl330 f8003000.ps7-dma: unable to set the seg size
3 [0.606344] dma-pl330 f8003000.ps7-dma: Loaded driver for PL330 DMAC-2364208
4 [0.613367] dma-pl330 f8003000.ps7-dma: DBUFF-128x8bytes Num_Chans-8 Num_Peri-4
      Num_Events-16
```

Figure 5.1: Kernel Boot Log.

In order to use the DMA, there is a need to write a combination of platform and character driver as ARM does not provide any driver for PS-DMA [47]. But Xilinx has provided driver support for PS-DMA but it is not desirable to use that for our software-hardware communication as it is not the optimized one. First a platform driver is developed and on top of it a character driver [37] is developed which does the transaction of data. In order to write platform driver there is a need to change the device tree since we have to write our own driver related device tree in order to

get the resource from DMA controller present in the platform. So the device tree has been modified by including the below lines,

```
overlay_dma@78000000 \{
burst-length = <0x4>;
dma-channel = <0x1>;
overlay-depth = <0x800>;
reg = <0x78000000 0x2000>;
compatible = "xlnx, overlay-dma";
\};
```

Figure 5.2: DTS file entry.

With respect to character driver, there are many file operations functions available in order to give abstraction APIs to the application. Out of those, the important operations used in our case is shown in Figure.5.3,

```
  struct file_operations mem_fops = {
            .open = mem_open,
            .release = mem_release,
            .read = mem_read,
            .write = mem_write,
  };
```

Figure 5.3: File operations.

From the operation names it is clear that each one of these APIs are abstracted and given to the user-application to send and receive the necessary data and all the experiments profiled the code present in write file operation which is our major focus.

5.2 DDR-DDR Communication

Before proceeding to test the data transfer between DDR and PL memory first, we started with the transfer between two different DDR memory locations. All the test are done using general purpose [GP] ports. There are basically two methods for PS-PL communication using GP ports -

- PIO [Programmed Input Output] wherein the processor itself involves in writing and reading of data
- Using PS-DMA wherein the processor will not involve in data transfer rather offload it to DMA present in the processor side.

With regard to OS like Linux, there is no straight way to access the data from DDR and put it to PL-BRAM since it is heavily abstracted from the application side. In Linux world, there are two spaces i) User-land or userspace where user does not have any privileged access to the underlying hardware but it can write any applications and ii) Kernel space which serves as a bridge between underlying platform devices and userspace. This has all privileges to access any device in the platform and their process priorities are higher than the userspace process.

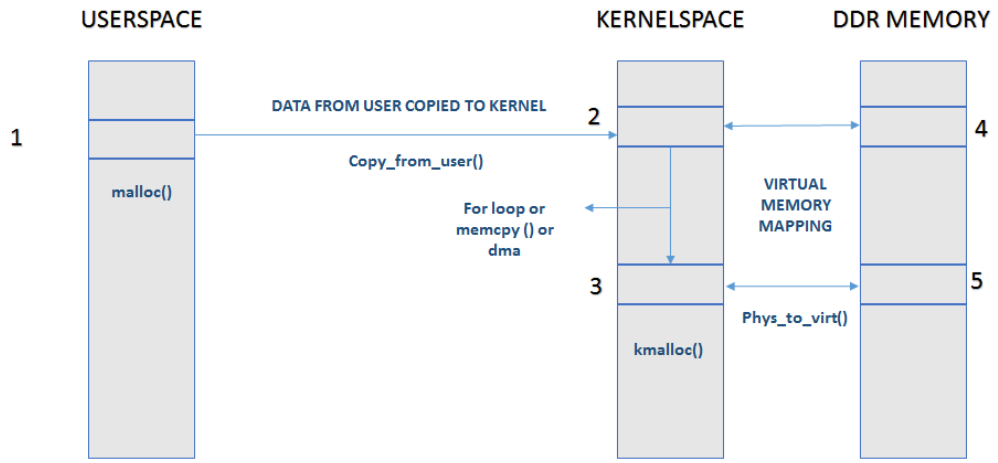

Figure 5.4: Data transaction flow

5.2.1 High level view of data transport

The overall picture of the experiment is given in Fig. 5.4 and explained as follows:

- 1. Indicates userspace virtual memory allocated using the library call malloc();
- 2. Indicates transfer of data from userspace to kernel virtual memory through copy_from_user kernel function call.

- 3. Kernel virtual memory is created by doing phys_to_virt() function call which performs the mapping of actual physical address to kernel virtual address which can be accessed by the kernel. This means writing to this kernel memory implies writing to the physical address. Copying of data from 2 to 3 can be done through either memcpy which is PIO or using PS-DMA.
- 4 & 5 indicates the actual physical address present in the system.

5.2.2 Non-DMA experiments

Table 5.1: DDR-DDR latency and bandwidth using PIO Userspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	73	1.7
64	73	3.5
128	73	7
256	73	14
512	73	28
1K	117	35
2K	171	48
4K	256	64
8K	442	74

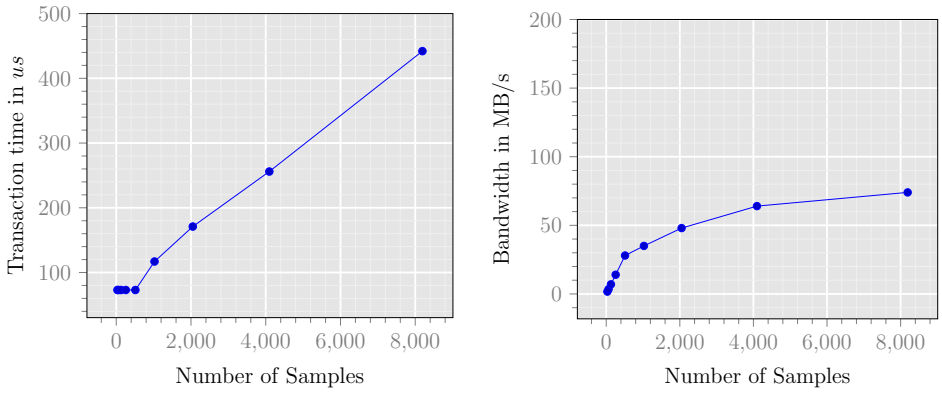

Figure 5.5: (a) userspacelatency_pio in us, and (b) userspacebw_pio in MB/s

Using PIO the measurement is taken in the userspace application code wherein the fwrite() function call is profiled using gettimeofday() function call. Since fwrite is the API provided by the driver this is used to send 32-bit samples for transfer. As observed from the Table 5.1 and Fig. 5.5(a), till 512 Samples the time measured is same and it increases as the number of samples increased. Fig. 5.5(b) shows the bandwidth in MB/s.

Table 5.2: DDR-DDR latency and bandwidth using PIO Kernel space Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	3	42.7
64	5	51.2
128	9	57
256	10	102.4
512	20	102.4
1K	36	113.7
2K	81	101.1
4K	164	100
8K	346	94.7

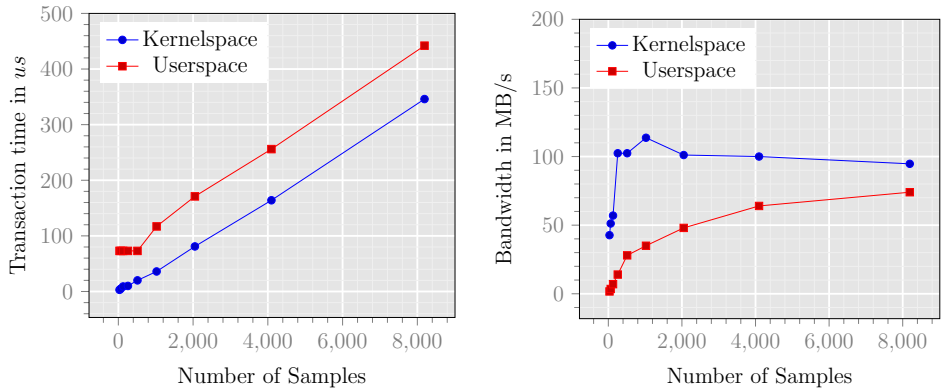

Figure 5.6: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for PIO userspace and kernelspace measurement

Table 5.2 provides values measured in the kernel space write API where it does copy the data from user through copy_from_user() and further copies the data to the intended DDR memory location through for loop. As observed from the Fig.5.6(b)

and Fig. 5.6(a), non-dma driver does not perform well (bandwidth is not going beyond $150MB/s$) when processor is used for copying of data. Hence, we develop PS-DMA driver which can improve the bandwidth beyond 150 MB/s. One key observation from Fig.5.6(a) is that the system call overhead is approximately 70 us.

5.2.3 PS-DMA experiments

Initially non-optimized PS-DMA driver is developed by considering only the functionality of the driver and initial bringing up of the platform.

```
1   dma_cap_set(DMA_SLAVE, mask);
2   /** here you have to get the dma channel */
3   /** First stage to get request slave channel*/
4   gp_dma_rx = dma_request_channel(mask, NULL, NULL);
5   if(!gp_dma_rx)
6   {
7   printk(KERN_INFO"So simple man... \n\r");
8   return ENOTSUPP; /* Just a fancy return value */
9   }
10  cookie = dma_async_memcpy_buf_to_buf(gp_dma_rx, d_base, c_base,
        count);
11
12  while(dma_async_is_tx_complete(gp_dma_rx, cookie, NULL, NULL) ==
        DMA_IN_PROGRESS)
13  {
14  dma_sync_wait(gp_dma_rx, cookie); /** just return success */
15  }
16  tot_bytes = count;
17  if(gp_dma_rx)
18  {
19  dmaengine_terminate_all(gp_dma_rx);
20  dma_release_channel(gp_dma_rx);
21  }
22  return count;
23  }
```

Figure 5.7: mem-write-call

The main part of the code which is being profiled in the entire part of the experiment is write API present in the driver code and code snippet attached in the Fig.5.7.

Results for userspace measurement got for this driver can be observed in the table 5.3

Table 5.3: DDR-DDR latency and bandwidth using PS-DMA Userspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	74	1.7
64	74	3.5
128	74	7
256	74	14
512	74	27.6
1K	135	30.3
2K	156.5	52.3
4K	212	77.3
8K	324.5	101

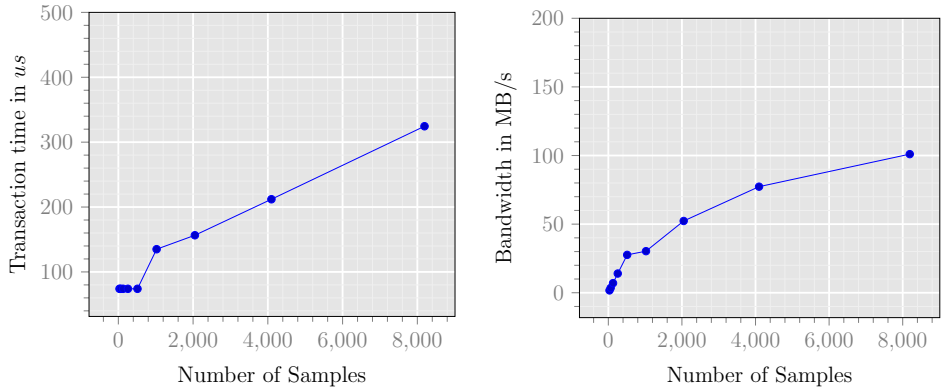

Figure 5.8: (a) userspacelatency_psdma in us, and (b) userspacebw_psdma in MB/s

From the Fig. 5.8(a) and Fig. 5.8(b), it is observed that the userspace measurement performs slightly better than PIO method at higher number of samples and it can able to achieve around 100 MB/s. The reason of better performance can be

Table 5.4: DDR-DDR latency and bandwidth using PS-DMA Kernel space Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	38.5	3.3
64	37	7
128	42	12.2
256	40	25.6
512	57	36
1K	52	78.7
2K	77	106.4
4K	127	129
8K	227	144

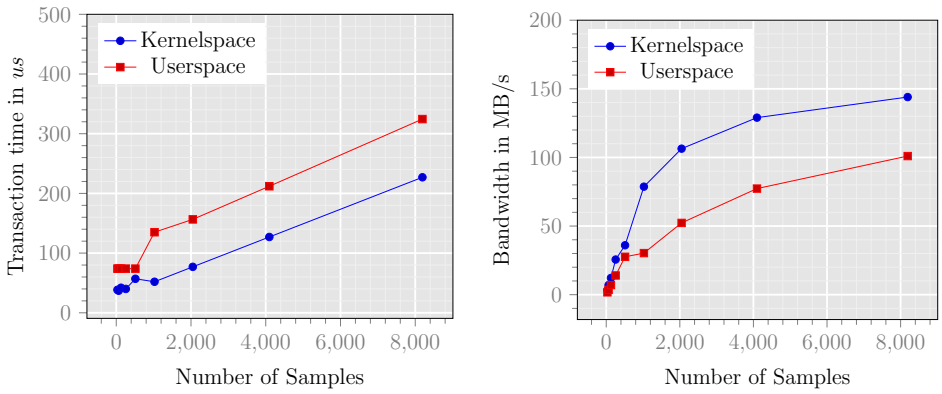

Figure 5.9: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement - psdma

the use of DMA transfer instead of memcpy API. It's actually a point of discussion on when to use DMA for data transfer versus just using memcpy. In case of sending a large amount of samples (may be greater than 1K), the benefit of using DMA increases significantly compared to memcpy.

From the Fig. 5.9(a) and Fig. 5.9(b) it is observed that the PS-DMA driver slightly better performance in both kernel and userspace compared to memcpy API. The achievable bandwidth is close to 150MB/s when we are using a basic driver implementation of PS-DMA. The performance can be improved using driver optimizations

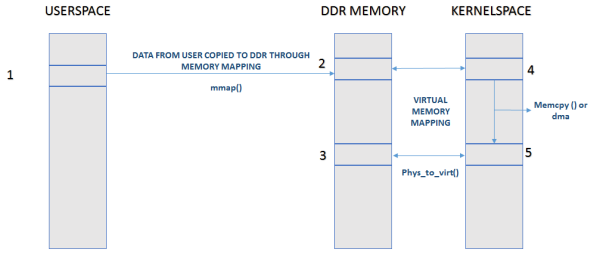

Figure 5.10: MMAP data transfer flow.

such as moving DMA initialization and releasing routines to the open and release driver function. Such few optimizations provide us a head-start towards performance improvement of PS-DMA driver.

5.3 Driver Engineering

Next we will move on to the optimized PS-DMA driver code in which following things are done:

- Moving from dma_aysnc_memcpy_buf_to_buf () to prep_dma_memcpy () makes a huge difference since the former depends on the cookie completion status whereas the latter depends on the interrupt.
- Implementation of mmap() system call in the userspace application code. This requires two things to be developed. First is the application has to know the physical DDR memory address which the kernel will be using for copying the data from userspace. Secondly the /dev/mem pipe, which represents physical DDR address, is used for doing mmap and getting a virtual address.
- Moving the dma channel initialization and releasing portions to open and release driver function calls which improves the write and read driver function calls. Initially, it was present as part of write and read file operation in the driver which results in higher latency since a part of the time is given for initialization and releasing operations.

Optimized driver code snippet can be seen in the Fig.5.11.

```
/**Getting the transcriptor*/
p_txdma_desc = gp_dma_tx->device->device_prep_dma_memcpy(
    gp_dma_tx, DDR_END,
DDR_START, count, DMA_PREP_INTERRUPT);

/** Initializing callbacks*/
p_txdma_desc->callback = &rxd_dma_callback;
p_txdma_desc->callback_param = NULL;
dmaengine_submit(p_txdma_desc);
/** Final stage to issue pending signal*/
dma_async_issue_pending(gp_dma_tx);
while(!gdma_check)
{
/** Checking whether the CDMA goes back to idle or not */
printk("received \n\r");
}
```

Figure 5.11: optimized-mem-write-call

Table 5.5: DDR-DDR latency and bandwidth using optimized PS-DMA Userspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	62.6	2
64	73.6	3.5
128	73	7
256	99.6	10.3
512	114.6	18
1K	95.8	42.7
2K	114.2	71.7
4K	114.2	143.4
8K	118.2	277.2
16K	147.4	444.6

Fig.5.12(a) and 5.12(b) shows the userspace measurement graph and the values are captured in the table 5.5.

Fig.5.13(a) and 5.13(b) shows the kernelspace measurement graph and the values are captured in the table 5.6.

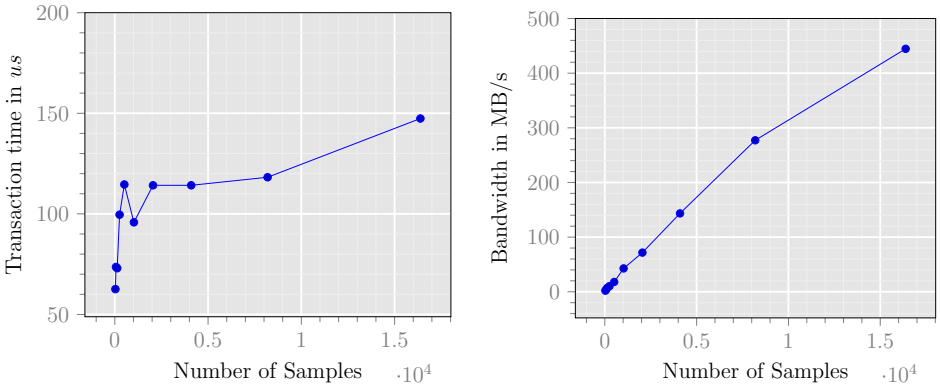

Figure 5.12: (a) userlatency_psdmammapddr in us, and (b) userbw_psdmammapddr in MB/s

Table 5.6: DDR-DDR latency and bandwidth using PS-DMA Kernel space Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	37	3.5
64	47	5.4
128	47.6	10.7
256	48	21.3
512	55	37.4
1K	41	100
2K	44	186.2
4K	55.2	297
8K	29	555.4
16K	92	712.3

It is clearly observed this optimization gives a lot better performance than the non-optimized PS-DMA driver. It is found that optimized DMA driver is approximately has bandwidth of about 4.5× times better than the non-optimized DMA driver and 6× times better than PIO driver in the kernelspace measurement.

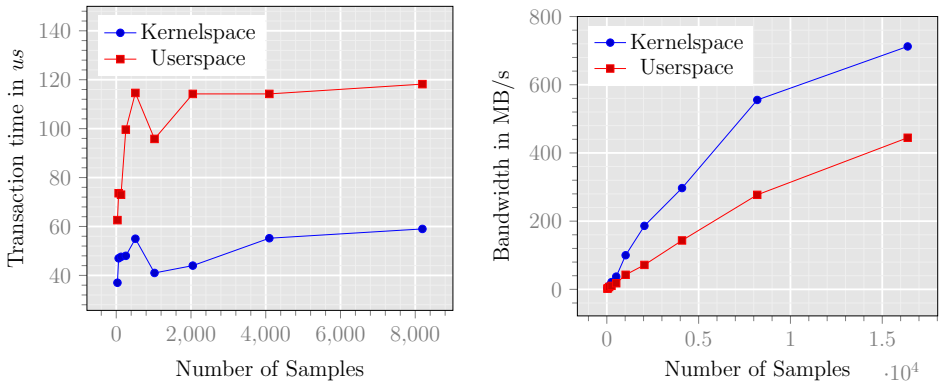

Figure 5.13: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement - mmap psdma

5.4 DDR-PL Communication

In the same way as DDR-DDR driver, the DDR-PL driver is developed in an optimized way as mentioned in the previous driver engineering section but the difference lies in the specification of destination addresses that the DMA will work with.

Table 5.7: DDR-PL latency and bandwidth using optimized PS-DMA Userspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
64	77.6	3.3
128	73.6	7
256	92	11
512	114.5	18
1K	105	39
2K	112.4	73
4K	142	115.4
8K	186.2	176
16K	284	230.7

In the case of hardware in the PL side, a physical BRAM is created that supports 16K 32-bit wide samples in order to do the experiments.

Fig 5.14(a) and 5.14(b) shows the userspace measurement graph and the values

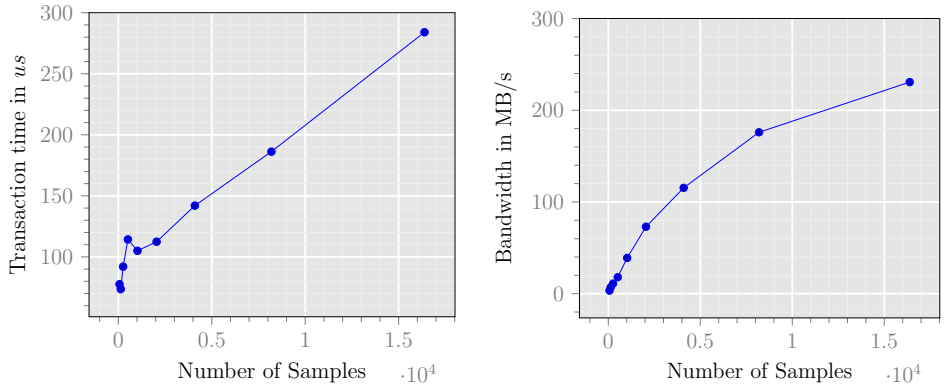

Figure 5.14: (a) userspacelatency_pl330_mmappl in us, and (b) userspacebw_pl330_mmappl in MB/s

are captured in the table 5.7.

Table 5.8: DDR-PL latency and bandwidth using optimized PS-DMA Kernel space Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
32	33.2	3.8
64	33.2	7.7
128	37	14
256	38.6	26.5
512	38.7	53
1K	38.7	106
2K	47.7	171.7
4K	74	221.4
8K	118	277.7
16K	219	303.4

Fig 5.15(a) and 5.15(b) shows the userspace measurement graph and the values are captured in the table 5.8.

As we know the maximum theoretical performance in case of PS-DMA driver transferring 32-bit samples to PL is 400 MB/s and our optimized driver can achieve approximately 300 MB/s for 16K samples data transfer.

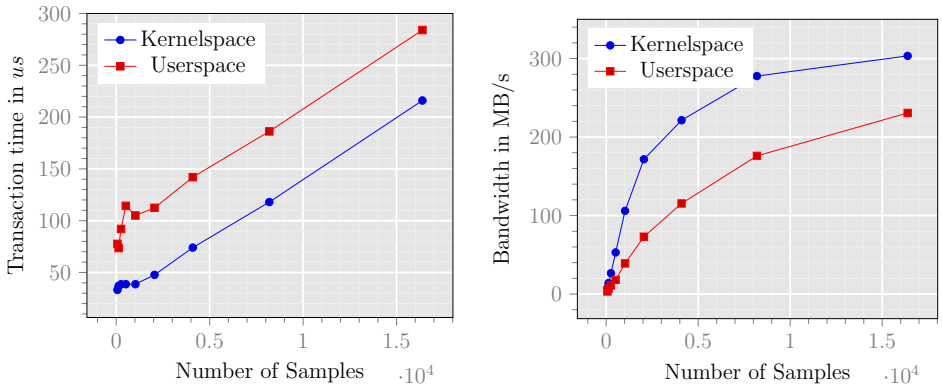

Figure 5.15: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement - mmap psdma PL transfer

5.5 Summary

In this chapter the characterization of data transactions between DDR-DDR memory using PIO and PS-DMA driver and DDR-BRAM using PS-DMA driver and also about the driver engineering wherein we discussed about the optimization done in order to achieve closer to the theoretical performance.

From the experiements we can conclude that, for DDR-DDR transactions 4.5× better than the non-optimized DMA driver and 6× times better than PIO driver in the kernelspace measurement. The difference between user-space and kernel-space performance is because of system call overhead which is approximately 70 us. For lesser number of samples (below 1K) non-DMA method performs well as DMA has some initialization overhead and makes it unfavorable for less number of samples. We used the optimized PS-DMA driver for DDR-PL transactions. We observe a throughput of up to 230 MB/s in user-space and up-to 300 MB/s (theoretical maximum bandwidth of 400 MB/s) in kernel space. The reduction in bandwidth in case of DDR-PL transactions is due to the operating frequency of memory sub-system which is 100 MHz. In future work, we plan to scale the frequency up to 250 MHz and observe the effect on achievable communication bandwidth.

Chapter 6

PL-DMA based Communication

6.1 Introduction

In order to enhance the transactions between DDR and PL, Xilinx has provided soft IP DMA cores that can achieve very high performance. The major aim for this kind of soft DMA is to ensure the communication latency between ARM and FPGA should be very minimal. Since by nature, FPGA is much faster in the execution and act as a co-processor it is heavily deployed in many applications. This potential can be fully used only by having minimal communication latency between ARM and FPGA. For this reason in case of zynq, Xilinx has provided three DMA IP cores which are [45],

- AXI-CDMA - Central Direct Memory Access
- AXI-DMA - Direct Memory Access
- AXI-VDMA - Video Direct Memory Access

Our major focus is on CDMA and DMA since the former does transaction between memory mapped regions while the latter does on memory map to stream region and vice-versa. AXI-VDMA is used in the case of video traffic related applications and it is not characterized in our work.

6.2 AXI Centralized DMA

CDMAs major application is in the area of transferring data between memory mapped source address and destination address. It provides a much higher bandwidth and it supports both 32-bit, 64-bit samples width with maximum burst size of 256. This can run at both 100 and 150 MHZ frequencies. This can be used to transfer data from DDR-DDR as well as DDR-BRAM [PL]. Xilinx has provided a platform driver for CDMA and it is already integrated in the Xillinux build. As mentioned already, it can be made either as an in-built or loadable module and for all the following experiments we use the driver as loadable module. Steps needed in order to make the CDMA working are as follows,

- 1. Generate bitstream with necessary hardware involving CDMA IP core instantiated in the Xilinx Platform Studio [XPS] and connecting a peripheral in order to store the data in it.
- 2. Two bitstreams are generated one with HP port and other with ACP port.
- 3. Make an entry in the device-tree file with the correct address in the generated in XPS. The entry currently made in the device-tree file is shown in Fig.6.1,

```
1  axicdma_0@axicdma40200000 {
2    compatible = "xlnx,axi-cdma";
3    reg = <0x40200000 0xFFFF>;
4    interrupt-parent = <0x2>;
5    interrupts = <0x0 0x3E 0x4>;
6    dma-channel@40200000 {
7       xlnx,device-id = <0x0>;
8       xlnx,datawidth = <0x40>;
9       interrupts = <0x0 0x1e 0x4>;
10   };
11 };
```

Figure 6.1: DTS entry for CDMA.

Code snippet for CDMA transfer is shown in the Fig.6.2. With the above setup and driver code the experiments are done for 64-bit transfer from DDR-DDR and then DDR-BRAM. Lets deal with DDR-DDR transfer by connecting the CDMA with one

```
1   /**Getting the transcriptor*/
2   p_txdma_desc = gp_dma_tx->device->device_prep_dma_memcpy(gp_dma_tx,
        DDR_DEST,
3   DDR_START, count, DMA_PREP_INTERRUPT);
4   p_txdma_desc->callback = NULL;
5   p_txdma_desc->callback_param = NULL;
6   /** Submit the obtained descriptor*/
7   dmaengine_submit(p_txdma_desc);
8   /** Final stage to issue pending signal*/
9   dma_async_issue_pending(gp_dma_tx);
10
11  while(((gdma_check = ioread32((VP)gp_cdma_sr)) & 0x0002 ) != C_VALUE)
12  {
13  /** Checking whether the CDMA goes back to idle or not */
14  }
```

Figure 6.2: CDMA-write-API

of the four HP ports first. The driver is optimized fully with mmap call present in the application and the obtained graphs are as follows,

6.2.1 DDR-DDR Communication

DDR-DDR with Single HP port - This deals with the measurement of latency and bandwidth in the user-space context using single HP port. An AXI-CDMA is instantiated in the hardware and it is connected to one of four HP ports and then the bitstream is generated. As mentioned earlier the devicetree file is changed according to the hardware and then experiments are done.

Fig 6.3(a) and 6.3(b) shows the userspace measurement graph and the values are captured in the table 6.1. Range taken is from 32 to 16K samples and the values for bandwidth can reach maximum of 800 MB/s but due to the overhead of system calls to the kernel space the measurement cannot above the kernel space bandwidth measurement.

Fig.6.4(a) and 6.4(b) shows the obtained kernel space measurement and the values are captured in the table 6.2

It is noted that the difference between the kernel and user space time is always constant in this case around 70 us which makes the bandwidth to be higher for kernel

Table 6.1: DDR-DDR Latency and bandwidth using AXI-CDMA Userspace Measurement through single HP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	73.7	3.5
64	74	7
128	90.3	11.3
256	112.3	18.2
512	85	48.2
1K	92	89
2K	110.6	148
4K	121.6	269.5
8K	166	394.7
16K	252.2	519.7

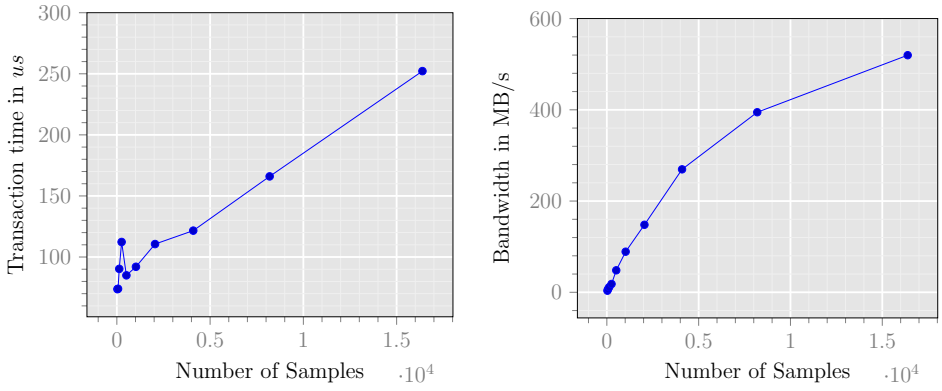

Figure 6.3: (a) userlatency_cdmaddrhp in us, and (b)userbw_cdmaddrhp in MB/s

than user as pointed out earlier. This is due to the face that the user application has to make several system calls to reach the kernel space which is the major reason for the higher latency and lower bandwidth for the userspace. Another important observation is the bandwidth obtained in the case of DDR-DDR transaction is much closer to the theoretical maximum of 800 MB/s.

DDR-DDR with ACP port - With regard to hardware, CDMA is being connected to ACP port rather than HP port and the driver, devicetree remains the same.

Table 6.2: DDR-DDR Latency and bandwidth using AXI-CDMA Kernel space Measurement using single HP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	18.4	14
64	18.2	28
128	20.3	50.4
256	24.2	84.6
512	22	186.2
1K	27.7	295.7
2K	37	443
4K	59	555.4
8K	98	668.7
16K	181	724

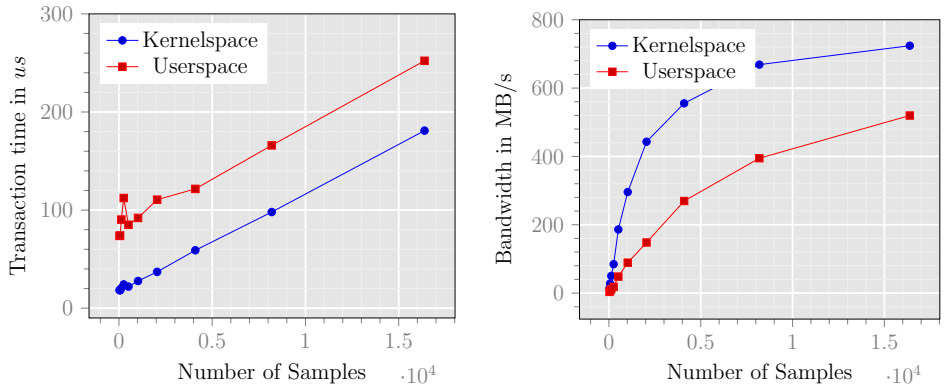

Figure 6.4: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using single HP port

Fig.6.5(a) and 6.5(b) shows the latency and bandwidth measurement in the user-space context and the values are captured in the table 6.3. It is clear that the userspace performance difference between HP and ACP ports for DDR transactions is very minimal.

Fig.6.6(a) and 6.6(b) shows the latency and bandwidth measurement in the kernel space and the values are captured in the table 6.4. It is observed that HP and ACP ports bandwidth performance are almost equal in the case of data transfer in kernel

Table 6.3: DDR-DDR Latency and bandwidth using AXI-CDMA Userspace Measurement through ACP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	73.8	3.5
64	77.8	6.6
128	85	12
256	110.6	18.5
512	88.4	46.4
1K	92	89
2K	110.8	148
4K	129	254
8K	158.4	413.7
16K	247.2	530.2

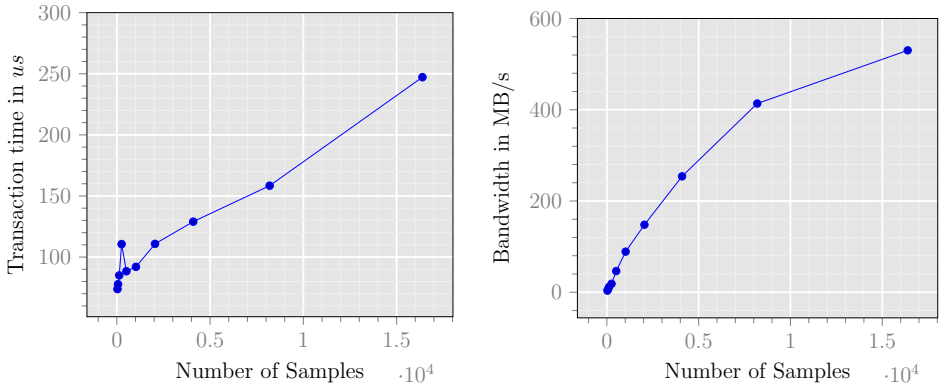

Figure 6.5: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace measurement using ACP port

space as well for DDR transactions and we can able to achieve close to the theoretical maximum of 800 MB/s.

DDR-DDR with Two HP ports:

In order to further increase the performance we tried to use the other HP ports with CDMA since it has four ports while in the case of ACP it is a single port so we cannot able to do much experiments with ACP port. Initially two CDMAs are instantiated to use HP0 and HP2 along with two peripherals using XPS in order to see the performance. With respect to this hardware change the device-tree is changed

Table 6.4: DDR-DDR Latency and bandwidth using AXI-CDMA Kernelspace Measurement through ACP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	18.4	14
64	18.2	28
128	18.2	56.3
256	18.4	111.3
512	18.6	220
1K	25.8	317.5
2K	36.6	447.6
4K	55.4	591.5
8K	92.2	711
16K	180.6	725.7

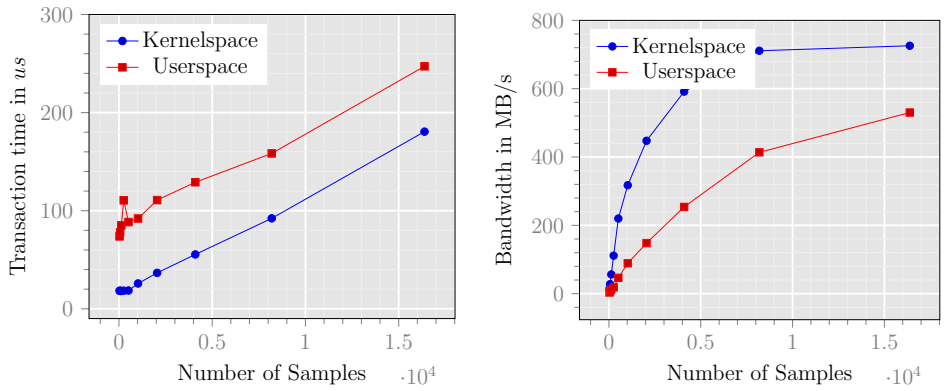

Figure 6.6: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using single HP port

in order to include two CDMA instantiations in the Fig.6.7,

Accordingly the driver is modified to get two slave channels from each CDMA present in the hardware and the procedure is followed for the two slave channels. The obtained graph for DDR-DDR transfer is shown in the Fig.6.8(a) and 6.8(b) in user-space context and its values are captured in the table 6.5 ,

It is observed that the two HP ports latency is quite high compare to single port as the driver waits for the two CDMA to get finished and hence the slight difference.

```
axicdma_0@axicdma40200000 {
compatible = "xlnx,axi-cdma";
reg = <0x40200000 0xFFFF>;
interrupt-parent = <0x2>;
interrupts = <0x0 0x3E 0x4>;
dma-channel@40200000 {
xlnx,device-id = <0x0>;
xlnx,datawidth = <0x40>;
interrupts = <0x0 0x1e 0x4>;
};
};

axicdma_1@axicdma40240000 {
compatible = "xlnx,axi-cdma";
reg = <0x40240000 0xFFFF>;
interrupt-parent = <0x2>;
interrupts = <0x0 0x3D 0x4>;
dma-channel@40200000{
xlnx,device-id = <0x1>;
xlnx,datawidth = <0x40>;
interrupts = <0x0 0x1d 0x4>;
};
};
```

Figure 6.7: DTS entry for CDMA - Two HP ports.

Table 6.5: DDR-DDR Latency and Bandwidth using AXI-CDMA Userspace measurement through two HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Two-ports Bandwidth in MB/s
16	73	1.7	3.4
32	73.6	3.5	7
64	81	6.3	12.6
128	86.7	11.8	23.6
256	112.4	18.2	36.4
512	92.2	44.4	89
1K	106.7	76.8	143.6
2K	110.7	148	296
4K	127	258	516
8K	175	374.5	750
16K	263.6	497	995

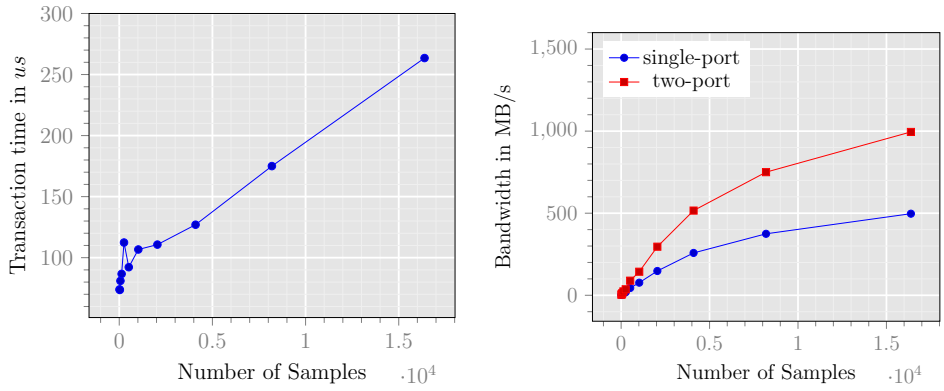

Figure 6.8: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace measurement one port vs two HP ports

This is reflected in the bandwidth achieved and but it is clear that the total bandwidth achieved is much greater than the single port as we are sending samples in two ports in parallel.

Table 6.6: DDR-DDR Latency and Bandwidth using AXI-CDMA Kernelspace Measurement through Two HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Two-ports Bandwidth in MB/s
16	27.7	4.6	9.2
32	26	10	20
64	29.5	17.3	34.6
128	29.6	34.6	69.2
256	31.5	65	130
512	24.5	167.2	334.4
1K	29.4	278.6	557.2
2K	46.2	354.6	709.2
4K	64.6	507.2	1014.4
8K	110.7	592	1184
16K	193.5	673.4	1347

The obtained kernel space measurement is shown in the Fig.6.9(a) and 6.9(b) and the values are captured in the table 6.6.

The key observations are, having a difference of approximately 70 us between userspace and kernel space latency, latency is quite high for two ports compare to

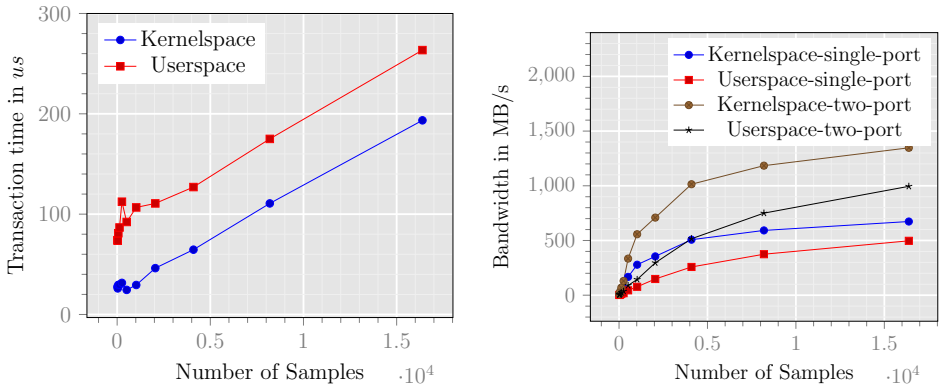

Figure 6.9: (a) The transaction time in us, (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using two HP ports vs one HP port

single port and the total bandwidth achieved is much higher than single port as expected.

6.2.2 DDR-PL Communication

With respect to DDR-PL BRAM transaction, device-tree and hardware remains same except for the driver code in which the destination address is changed according to the PL-BRAM's address generated in the XPS.

Table 6.7: DDR-PL Latency and bandwidth using AXI-CDMA Userspace Measurement through HP single port

Number of Samples	Latency in us	Bandwidth in MB/s
32	73.5	3.5
64	78	6.5
128	92	11
256	110	18.6
512	88	46.5
1K	92	89
2K	103	159
4K	122	268.5
8K	166	394

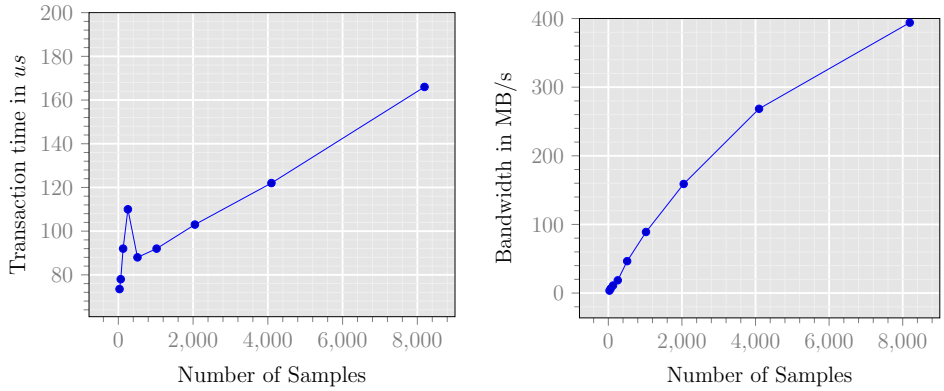

Figure 6.10: (a) userlatency_plcdmaHP in us, and (b)userbw_plcdmaHP in MB/s

In our experiments, we have used $0x6AA00000$ as the BRAM memory address since the hardware is generated with that address in XPS. Then we performed similar experiments as in DDR-DDR having both single and two ports performance measurement with respect to HP port.

Fig.6.10(a) and 6.10(b) shows the single HP port measurement in the userspace context and the values are captured in the table 6.7. It is observed that CDMA gives approximately equivalent performance for DDR-DDR and DDR-PL transactions in the case of userspace measurement.

Table 6.8: DDR-PL Latency and bandwidth using AXI-CDMA Kernel space Measurement through single HP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	15	17
64	18	28
128	20	51
256	24	85
512	18.5	221
1K	27	303
2K	37	443
4K	56	585
8K	96	669

Fig.6.11(a) and 6.11(b) shows the kernel space measurement and the values are

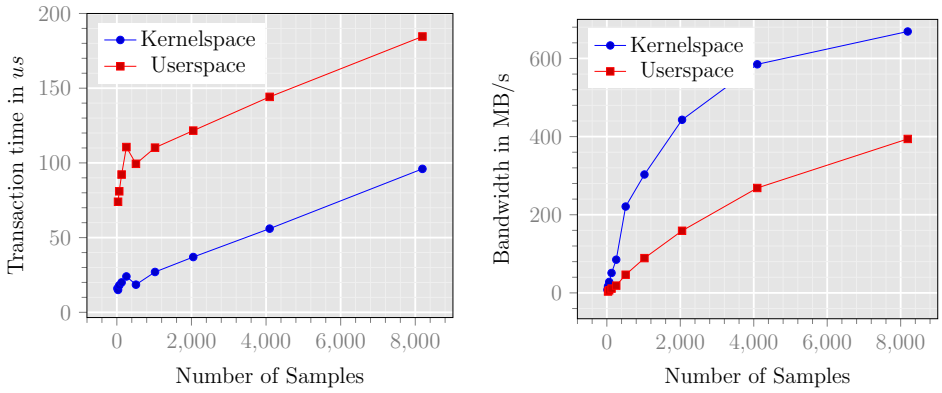

Figure 6.11: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using two HP ports DDR-PL

captured in the table 6.8

The main observations are, having a difference of approximately 70 us between userspace and kernel space latency as expected and the bandwidth achieved is approximately equivalent to that DDR-DDR transaction and it clearly indicates CDMA is giving same performance irrespective of the system memory locations.

DDR-PL with ACP port - With regard to hardware the CDMA is being connected to ACP

Table 6.9: DDR-PL Latency and bandwidth using AXI-CDMA Userspace Measurement through ACP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	74	2.7
64	78	6.5
128	90	11.4
256	112	18.3
512	90	45.5
1K	98	83.5
2K	103	159
4K	122	268.5
8K	166	394

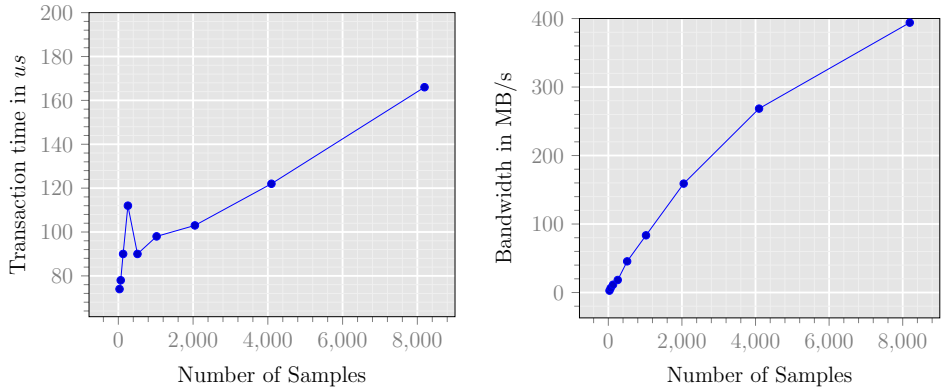

Figure 6.12: (a) userspacelatency_axicdma-acp-pl in us, and (b) userspacebw_axicdma-acp-pl in MB/s

port rather than HP port and the driver, device-tree remains the same. Fig.6.12(a) and 6.12(b) shows the userspace latency and bandwidth measurement and the values are captured in the table 6.9.

Table 6.10: DDR-PL Latency and bandwidth using AXI-CDMA Kernel space Measurement through ACP port

Number of Samples	Latency in us	Bandwidth in MB/s
32	16	15
64	16.5	31
128	18.4	55.6
256	18.4	111
512	20.4	201
1K	24	341
2K	37	443
4K	57	575
8K	100	655

It is observed that the latency and bandwidth are almost equivalent with HP port and it does not give much higher performance than the HP port. The bandwidth achieved at 8K samples is quite closer to the maximum theoretical bandwidth that can be achieved with CDMA through ACP port.

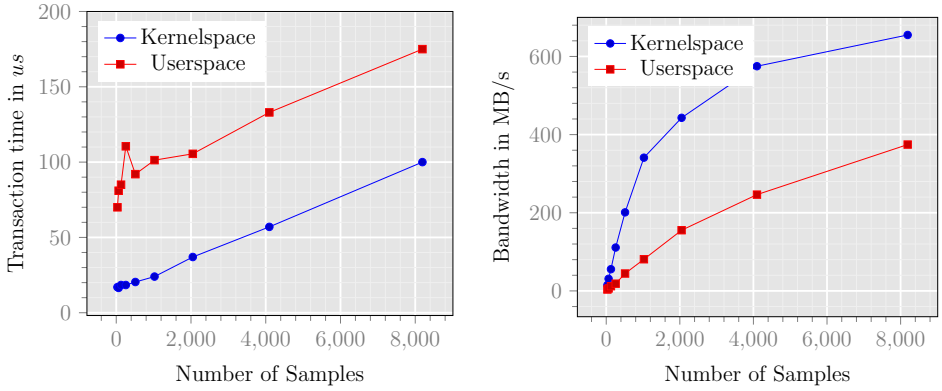

Figure 6.13: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using two HP ports DDR-PL ACP port

Fig.6.13(a) and 6.13(b) shows the kernel space measurement and the values are captured in the table 6.10.

The key observations are, having a difference of approximately 70 us between userspace and kernel space latency as expected and the bandwidth achieved is approximately equivalent to that DDR-DDR transaction and also to HP port. This clearly indicates CDMA is giving same performance irrespective of the system memory locations and also ports through which CDMA connects.

Table 6.11: DDR-PL Latency and Bandwidth using AXI-CDMA Userspace Measurement through Two HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Two-ports Bandwidth in MB/s
32	70	3.6	7.2
64	81	6.3	18.6
128	85	12	24
256	110.5	18.5	37
512	92	44.5	89
1K	101.3	81	162
2K	105.5	155.3	310.6
4K	133	246.4	493
8K	175	374.5	749

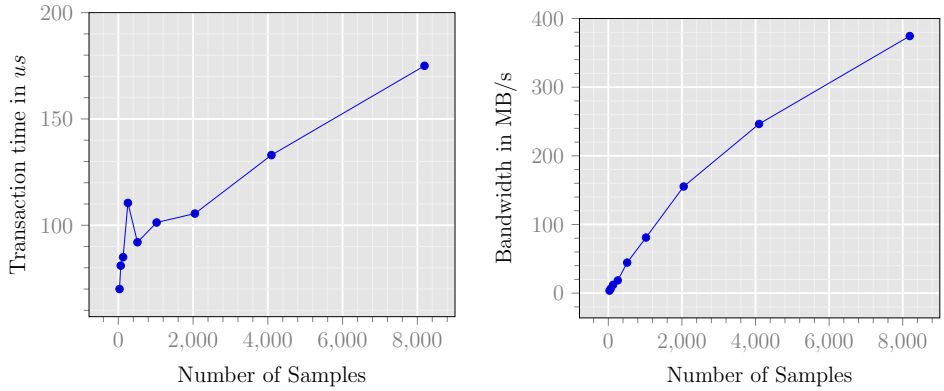

Figure 6.14: (a)userspacelatency_axicdma-twoports-pl in us, and (b) userspacebw_axicdma-twoports-pl in MB/s

DDR-PL with two HP ports - similarly we have done experiments using two ports of HP to see the performance.

Fig.6.14(a) and 6.14(b) shows the userspace latency and bandwidth measurement and the values are captured in the table 6.11.

Table 6.12: DDR-PL Latency and Bandwidth using AXI-CDMA Kernel space Measurement through Two HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Two-ports Bandwidth in MB/s
32	26	9.8	19.6
64	26	19.7	39.4
128	31.4	32.6	65.2
256	29.6	65.2	130.4
512	26	157.5	315
1K	35	234	468
2K	42.6	384.6	769
4K	62.6	523.5	1047
8K	103.2	635	1270

It is observed that the latency and bandwidth are almost equivalent with DDR-DDR transaction using HP ports. The bandwidth achieved at 8K samples is quite closer to the maximum theoretical bandwidth that can be achieved with CDMA.But

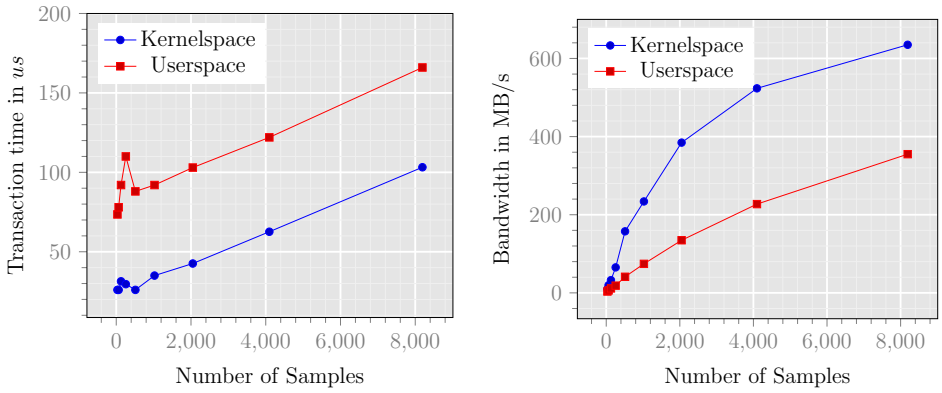

Figure 6.15: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement using two HP ports for DDR-PL transfer

as mentioned already, latency is a bit high when compared to single port HP experiment as the application has to wait for both the CDMAs to finish the transaction process.

Fig.6.15(a) and 6.15(b) shows the kernel space measurement and the values are captured in the table 6.12 and the observations are as same as two HP ports for DDR-DDR transaction.

After this we used all the four HP ports and try to do DDR-PL transaction for the maximum performance to attain between ARM and FPGA. Steps followed are similar as in for two ports instantiating four CDMA along with four peripherals, editing the driver code to get four slave channels and performing transactions for all the channels and also changing the device-tree file as shown in the Fig.6.16.

With these changes the experiments are done in the case of DDR to four BRAM locations which are obtained in the XPS and the observations are captured as follows.

Fig.6.17(a) and 6.17(b) shows the userspace latency and bandwidth measurement and the values are captured in the table 6.13.

As observed, the latency for doing the transaction is bit higher compare to two-ports HP port as the application has to wait for all the four CDMA to complete the transaction and hence the bandwidth also gets reflected. The total bandwidth achieved is much greater than the bandwidth of two HP ports and this is the maximum

```
1
2   axicdma_0@axicdma40200000 {
3   compatible = "xlnx,axi-cdma";
4   reg = <0x40200000 0xFFFF>;
5   interrupt-parent = <0x2>;
6   interrupts = <0x0 0x3E 0x4>;
7   dma-channel@40200000 {
8   xlnx,device-id = <0x0>;
9   xlnx,datawidth = <0x40>;
10  interrupts = <0x0 0x1e 0x4>;
11  };
12  };
13
14  axicdma_1@axicdma40240000 {
15  compatible = "xlnx,axi-cdma";
16  reg = <0x40240000 0xFFFF>;
17  interrupt-parent = <0x2>;
18  interrupts = <0x0 0x3D 0x4>;
19  dma-channel@40200000{
20  xlnx,device-id = <0x1>;
21  xlnx,datawidth = <0x40>;
22  interrupts = <0x0 0x1d 0x4>;
23  };
24  };
25  axicdma_2@axicdma40220000 {
26     compatible = "xlnx,axi-cdma";
27     reg = <0x40220000 0xFFFF>;
28     interrupt-parent = <0x2>;
29     interrupts = <0x0 0x3C 0x4>;
30     dma-channel@40220000{
31        xlnx,device-id = <0x2>;
32        xlnx,datawidth = <0x40>;
33        interrupts = <0x0 0x1c 0x4>;
34     };
35  };
36
37  axicdma_3@axicdma40280000 {
38     compatible = "xlnx,axi-cdma";
39     reg = <0x40280000 0xFFFF>;
40     interrupt-parent = <0x2>;
41     interrupts = <0x0 0x3B 0x4>;
42     dma-channel@40280000{
43        xlnx,device-id = <0x3>;
44        xlnx,datawidth = <0x40>;
45        interrupts = <0x0 0x1b 0x4>;
46     };
47  };
```

Figure 6.16: DTS entry for CDMA - Four HP ports.

that we could achieve with the setup.

Fig.6.18(a) and 6.18(b) shows the kernel space measurement and the values are captured in the table 6.14 and the observations are as same as two HP ports for DDR-DDR transaction.

The key observations are, having a difference of approximately 70 us between userspace and kernel space latency as expected and the bandwidth achieved is the highest with four CDMAs connected to four HP ports.

This clearly indicates usage of CDMA in high-performance applications which provide four simultaneous transfer of 555 MB/s per port which corresponds to the total of 2.2 GB/s.

Table 6.13: DDR-PL Latency and Bandwidth using AXI-CDMA Userspace Measurement through Four HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Four-ports Bandwidth in MB/s
32	74	3.5	14
64	81	6.3	25.2
128	92.2	11.4	44.4
256	110.6	18.5	74
512	99.4	41.2	165
1K	110.2	74.3	297.2
2K	121.6	134.7	539
4K	144.2	227.2	909
8K	184.6	355	1420

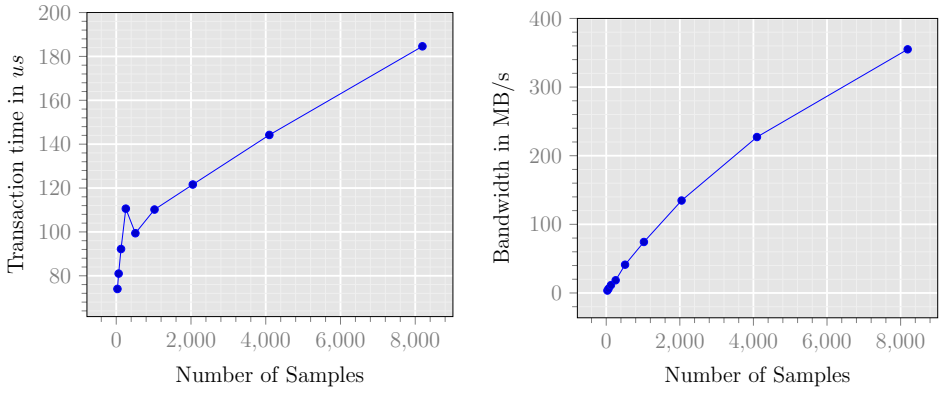

Figure 6.17: (a) userspacelatency_axicdma-four-ports in us, and (b) userspacebw_axicdma-four-ports in MB/s

After performing these experiments with single, two and four ports, the summarized graph in user and kernel spaces for bandwid is shown in the Figure 6.19 and 6.20 respectively.

It clearly says that, as the number of channels increases the latency also increases since the driver will wait for all the CDMA operations to be finished in order to make sure the functionality of the CDMA. This is the reason for the increase in latency and it is observed in both user and kernel space.

Table 6.14: DDR-PL Latency and Bandwidth using AXI-CDMA Kernel space Measurement through Four HP ports

Number of Samples	Latency in us	One-port Bandwidth in MB/s	Four-ports Bandwidth in MB/s
32	37	7	28
64	44	11	46.5
128	40.6	25.2	101
256	47.4	43.2	173
512	37	110.7	443
1K	40.6	202	808
2K	55.2	297	1188
4K	73.6	445.2	1781
8K	118	555.4	2221

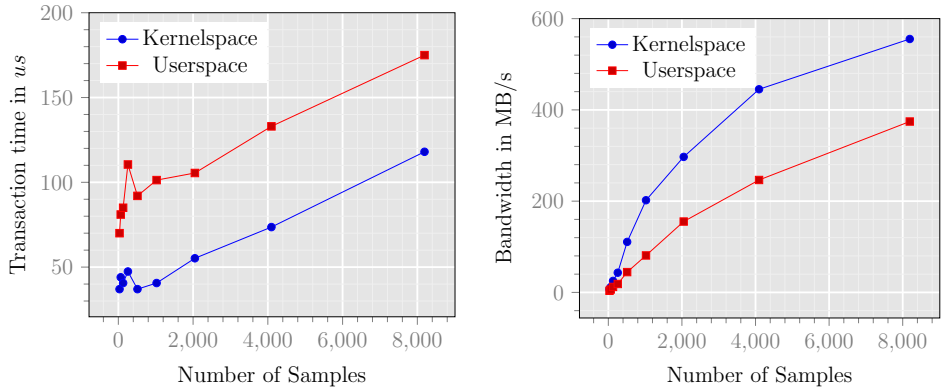

Figure 6.18: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement for four HP ports PL transfer

6.3 AXI DMA

The main application for AXI-DMA is in the case of streaming since it has the capability of converting memory mapped data to streaming data to the FPGA and get the streaming data back from FPGA convert it to memory mapped data in the ARM side.

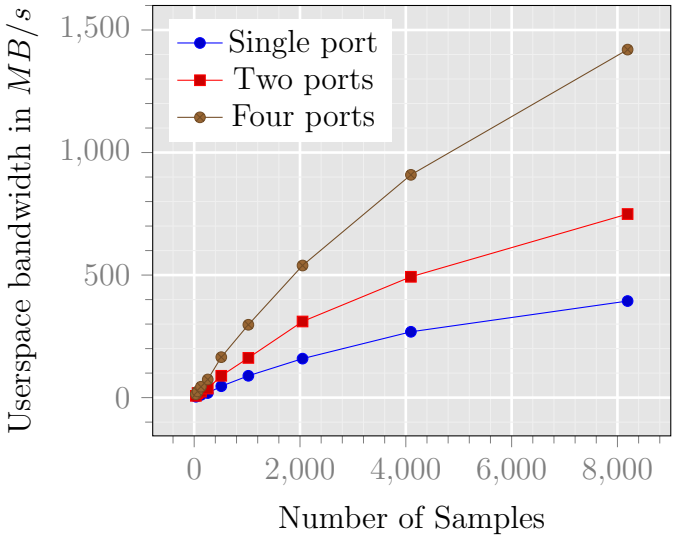

Figure 6.19: Userspace bandwidth in MB/s

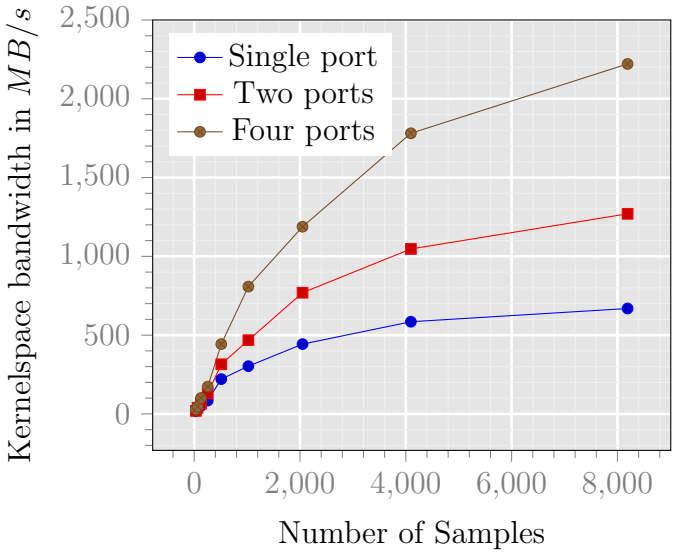

Figure 6.20: Kernelspace bandwidth in MB/s

The experiments are carried out in a loopback fashion in order to see the performance of DMA. Connecting the MM2S channel to the S2MM channel directly without any intermediate fabric gives the loopback hardware and all the experiments are carried out. The key changes done for this DMA are - device-tree file modified as

```
axidma\_0@axidma40400000 \{
compatible = "xlnx,axi-dma";
reg = <0x40400000 0xFFFF>;
interrupt-parent = <0x2>;
interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4>;
\#dma\_cells = <0x1>;
dma-channel@40400000 \{
xlnx,device-id = <0x0>;
xlnx,datawidth = <0x40>;
interrupts = <0x0 0x1e 0x4>;
compatible = xlnx,axi-dma-s2mm-channel;
};
dma-channel@40400030 {
xlnx,device-id = <0x0>;
xlnx,datawidth = <0x40>;
interrupts = <0x0 0x1e 0x4>;
compatible = xlnx,axi-dma-mm2s-channel;
};
};
```

Figure 6.21: DTS entry for AXI-DMA

```
/** Transfer the ownership to the dma handle and then take it after work*/
g_txdma_handle = dma_map_single(gp_dmadevtx, (VP)ddr_start, count,
    DMA_TO_DEVICE);
/**Getting the transcriptor*/
p_txdma_desc = dmaengine_prep_slave_single(gp_dma_tx, g_txdma_handle,
count, DMA_MEM_TO_DEV, DMA_PREP_INTERRUPT);
p_txdma_desc->callback = &txd_dma_callback;
p_txdma_desc->callback_param = NULL;
/** Submit the obtained descriptor*/
dmaengine_submit(p_txdma_desc);
/** Final stage to issue pending signal*/
dma_async_issue_pending(gp_dma_tx);
dma_unmap_single(gp_dmadevtx, g_txdma_handle, count, DMA_TO_DEVICE);
```

Figure 6.22: AXI-DMA Write API

shown in the Fig.6.21, driver code changes are mentioned in the section 4. The main function we will be profiling is the write API function call as shown in the Fig.6.22 and its equivalent fwrite function in the userspace.

Hardware is generated with axi-dma interface connected to single HP port and experiments carried out. Figure 6.23(a) and 6.23(b) shows the bidirectional latency

and bandwidth measurement in the user-space and the values are captured in the table 6.15,

Table 6.15: Bidirectional latency and bandwidth AXI-DMA - Userspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
512	151	27
1K	179	45.8
2K	236	69.5
4K	348	94
8K	599	109.5
16K	1065.5	121.3
32K	2040.5	128.5
64K	3793.2	138.2

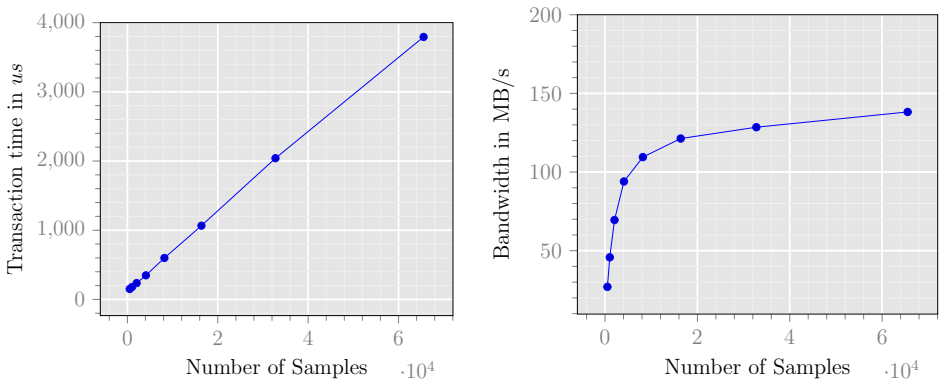

Figure 6.23: (a) userspacelatency_axidma-loopback in us, and (b)userspacebw_axidma-loopback in MB/s

Fig.6.24(a) and 6.24(b) shows the kernel space measurement and the values are captured in the table 6.16. It can be observed that the bidirectional bandwidth is saturating at around 140 MB/s.

The reason for doing loopback experiments with AXI-DMA is that the maximum burst size or the samples that the channel can hold is 256 corresponds to 2 KB of memory. If the samples are more than 256 then the DMA channel hangs since there

Table 6.16: Bidirectional latency and bandwidth AXI-DMA- Kernelspace Measurement

Number of Samples	Latency in us	Bandwidth in MB/s
512	86.6	47.3
1K	114.2	71.8
2K	166	98.7
4K	287.5	114
8K	520	126
16K	978.7	134
32K	1920.7	136.5
64K	3773.5	140

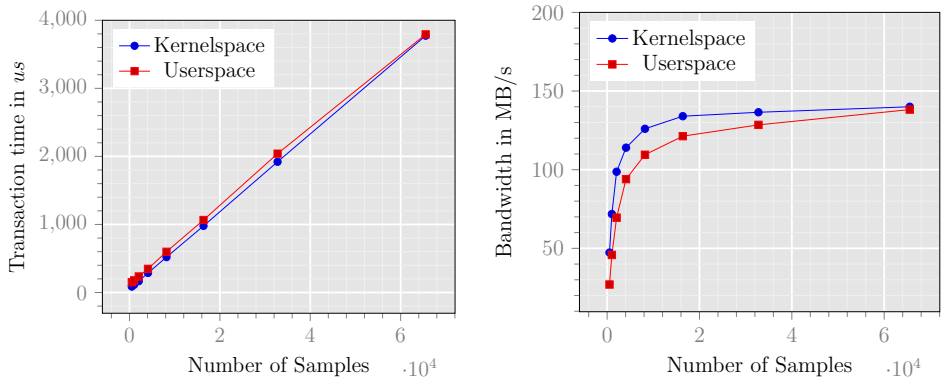

Figure 6.24: (a) The transaction time in us, and (b) The bandwidth in MB/s, vs the amount of data transferred for userspace and kernelspace measurement for DMA loopback

is a need to get the data back to some memory location. This clearly implies that we have to start receiving of samples through S2MM channel right after the MM2S channel is started fetching the samples. For this reason, the driver code for doing this transaction is present in the same write API of driver code.

After this experiment with HP port, we moved to carry out experiments with ACP port and the latency and bandwidth is found to be equivalent.

6.3.1 Comparison with Xillybus

In order to compare our results with the existing commercial infrastructure we chose Xillybus.

Table 6.17: Bidirectional Latency and bandwidth Using Xillybus

Number of Samples	Latency in us	Bandwidth in MB/s
1K	3544.4	1.15
2K	4676.3	1.75
4K	4394	3.7
8K	3721.5	8.8
16K	5104	12.8
32K	2706	48.4
64K	4639.5	56.5
128K	3367.5	155.7

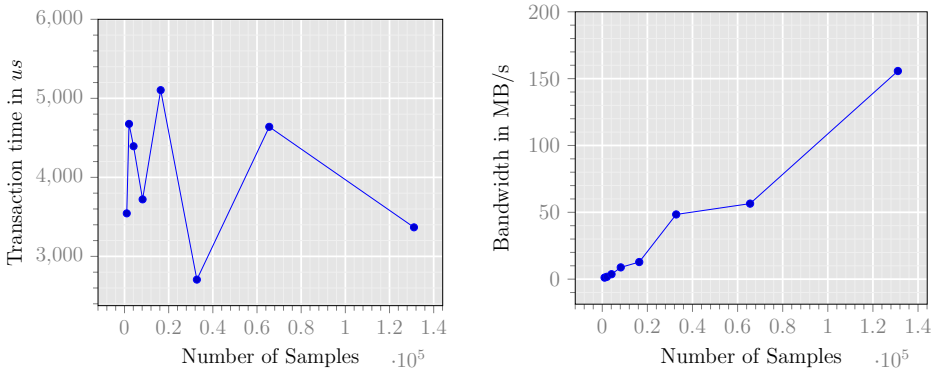

Figure 6.25: (a) userspacelatency_xillybus-loopback in us, and (b)userspacebw_xillybus-loopback in MB/s

It provides an infrastructure in which 32-bit samples can be sent to the underlying FPGA hardware through application FIFO's and get it back in an efficient manner as mentioned in the section 2. They have this hardware connected through AXI-DMA through ACP port and they have two pipes "/dev/xillybus_write_32" and "/dev/xillybus_read_32" through which the samples are sent and received.

From the Figures.6.25(a) and 6.25(b), it is clear that the commercially available Xillybus infrastructure does not perform well at all samples and the bandwidth is saturating at 150 MB/s at 128K samples and we can able to achieve 140 MB/s

at 64K Samples. From this, we can observe that our performance is comparable to commercially available infrastructure.

6.4 Summary

An analysis of PL-DMA driver and its performance when used for DDR-DDR transaction as well as for DDR-PL transaction and comparison of HP, ACP ports were presented in this chapter. The maximum theoretical performance of using AXI-DMA or AXI-CDMA with the single port is 800 MB/s supporting 64-bit data transfer. Using AXI-CDMA driver, we are able to achieve up to 730 MB/s (kernel-space) and up to 530 MB/s (user-space) in case of DDR-DDR transaction and up to 700 MB/s (kernel-space) and up to 400 MB/s (user-space) in case of DDR-PL transaction. The maximum performance achieved when connecting four HP ports with four CDMA is 1.4 GB/s in user space and 2.2 GB/s in kernel space against the theoretical value of 3.2 GB/s. The difference between user-space and kernel-space performance is because of system call overhead which is approximately 70 us as same as PS-DMA case.

The round trip bandwidth achieved in the case of our AXI-DMA driver is $140MB/s$ which is comparable to a commercially available system-level driver, Xillybus. The performance of this Xillybus driver were observed with the given kernel and presented and compared against the AXI-DMA driver that we have developed. The latency and bandwidth of our driver is better than Xillybus at some samples and almost comparable at higher samples such as 64K samples.

Chapter 7

Conclusions and Future Work

This chapter concludes and summarizes this report and in addition to it, we discuss future research directions in detail.

7.1 Conclusions

This report proposed an approach for sending data between DDR to DDR memory locations and DDR to BRAM locations in an optimized way achieving bandwidth close to theoretical maximum. Use of optimized DMA engine driver calls and moving the dma initialization overhead out of the write and read functionalities provided a bigger improvement in achieving a good bandwidth. This work consists of development of platform and character driver in case of PS-DMA and character driver in case of PL-DMA. Necessary hardware architecture using a memory subsystem is also created in order to establish a proper communication between PS and PL. This work concentrated on developing a driver that meets the standard rules of driver development and also optimized driver in which new technique such as Zero-copy is implemented. On the FPGA fabric, we developed an AXI-compliant, lightweight memory sub-system and on the processor, we develop Linux based drivers, specifically DMA drivers, to provide communication APIs. The proposed memory sub-system serves as a portable bridge between the accelerators and the external memory. Experiments were designed

to characterize software-hardware communication interfaces on the Xilinx Zynq platform within a general purpose operating system (Linux) framework to study the effect of interface choice on the maximum performance of these interfaces. We also measured the performance of developed PS-DMA drivers, PL-DMA drivers and provided a comparison against the commercially available system-level driver, Xillybus.

An analysis of PS-DMA driver and its performance when used for DDR-DDR transaction as well as for DDR-BRAM transaction and comparison with normal PIO method were presented in chapter 5. For DDR-DDR transactions (non-DMA), we observe a throughput of up to 80 MB/s in user-space and up-to 100 MB/s in kernel space. Using PS-DMA driver (un-optimized), we observe a throughput of up to 100 MB/s in user-space and up-to 150 MB/s in kernel space. Using Optimized PS-DMA driver, we observe a throughput of up to 450 MB/s in user-space and up-to 700 MB/s in kernel space. Hence, for DDR-DDR transactions the optimized PS-DMA driver performs 4.5× better than non-optimized PS-DMA driver and 6× better than non-DMA method. The difference between user-space and kernel-space performance is because of system call overhead which is approximately 70 us. For lesser number of samples (below 1K) non-DMA method performs well as DMA has some initialization overhead and makes it unfavorable for lesser number of samples. We used the optimized PS-DMA driver for DDR-PL transactions. We observe a throughput of up to 230 MB/s in user-space and up-to 300 MB/s (theoretical maximum bandwidth of 400 MB/s) in kernel space. The reduction in bandwidth in case of DDR-PL transactions is due to the operating frequency of memory sub-system which is 100 MHz. In future work, we plan to scale the frequency up to 250 MHz and observe the effect on achievable communication bandwidth.

An analysis of PL-DMA driver and its performance when used for DDR-DDR transaction as well as for DDR-BRAM transaction and comparison of HP, ACP ports were presented in chapter 6. The maximum theoretical performance of using AXI-DMA or AXI-CDMA with the single port is 800 MB/s with 64-bit data transfer. Using AXI-CDMA driver, we are able to achieve up to 730 MB/s (kernel-space) and up to 530 MB/s (user-space) in case of DDR-DDR transaction and up to 700 MB/s (kernel-space) and up to 400 MB/s (user-space) in case of DDR-PL transaction. The

maximum performance achieved when connecting four HP ports with four CDMA is 1.4 GB/s in user space and 2.2 GB/s in kernel space against the theoretical value of 3.2 GB/s. The difference between user-space and kernel-space performance is because of system call overhead which is approximately 70 us as same as PS-DMA case.

The round trip bandwidth achieved in the case of our AXI-DMA driver is $140MB/s$ which is comparable to the commercially available system-level driver, Xillybus. The performance of Xillybus were observed with the given kernel and presented in chapter 6 and compared against the AXI-DMA driver that we have developed. The latency and bandwidth of our driver is better than Xillybus at some samples and almost comparable at higher samples such as 64K samples.

7.2 Future work

Some of the main future research directions are developing optimized Linux DMA driver with minimum latency, overhead and maximum bandwidth for data transfer between main memory and accelerator memory. We describe these directions in detail as follows:

- **Improvement of PS-DMA driver**: The current platform driver utilized one of the eight PS-DMA channels which can be further extended by utilizing the remaining seven channels to see if performance can be improved further.
- **Improvement of AXI Centralized DMA (CDMA) driver**: The current CDMA character driver can be improved by introducing kernel threads in the driver to access the four HP ports in parallel which might increase the overall performance in terms of bandwidth.
- **Improvement of AXI DMA driver**: The current driver is a very generic and portable and it is not intended for high-performance applications. For applications that do not need driver portability those situations it can be developed using direct addressing of dma registers to give maximum performance.
- **Increase the PL frequency**: All the experiments done in this work uses default frequency of 100 MHz in PL. In future work, we plan to scale the

frequency up to 250 MHz and observe the effect on achievable communication bandwidth.

- **Measuring the power**: Measurement of power when running driver along with an application can be performed to provide a performance metric of power consumption.

Finally, with these initiatives we hope to develop a system-level driver (optimized for low latency and high bandwidth) for communication abstraction while performing data transactions between processor and streaming accelerators (implemented on reconfigurable fabric) on the Xilinx Zynq platform.

Bibliography

[1] ARM Ltd. The ARM Cortex-A9 Processors. http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[2] Xilinx Ltd. Zynq-7000 technical reference manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2013.

[3] Xillybus Ltd. Xillybus: IP Core Product Brief. http://xillybus.com/downloads/xillybus_product_brief.pdf.

[4] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[5] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL design and verification: a prescription for electronic system level methodology*. Morgan Kaufmann, 2010.

[6] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[7] http://www.eejournal.com/archives/articles/20130305-fpgawars/.

[8] https://en.wikipedia.org/wiki/Reconfigurable_computing.

[9] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 225–235, 2000.

[10] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 97–106, 2011.

[11] Khoa Dang Pham, Abhishek Kumar Jain, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013.

[12] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

[13] Xilinx Ltd. AXI Reference Guide. `http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf`.

[14] A. Agne, M. Platzner, and E. Lubbers. Memory virtualization for multithreaded reconfigurable hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 185–188, September 2011.

[15] Shinya Takamaeda-Yamazaki, Kenji Kise, and James C. Hoe. Pycoram: Yet another implementation of coram memory architecture for modern fpga-based computing. In *Workshop on the Intersections of Computer Architecture and Recon?gurable Logic (CARL)*, 2013.

[16] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336. 1996.

[17] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, November 2004.

[18] M. Vuletic, L. Righetti, L. Pozzi, and P. Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 748–749, 2004.

[19] K. Rupnow. Operating system management of reconfigurable hardware computing systems. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 477–478, 2009.

[20] Ivan Gonzalez and Sergio Lopez-Buedo. Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture. *Journal of Systems Architecture*, 58(6–7):247–256, June 2012.

[21] Enno Lübbers and Marco Platzner. ReconOS: multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, October 2009.

[22] H.K.-H. So, A. Tkachenko, and R. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 259–264, 2006.

[23] K. Kosciuszkiewicz, F. Morgan, and K. Kepa. Run-time management of reconfigurable hardware tasks using embedded linux. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 209–215, 2007.

[24] D. Gohringer, S. Werner, M. Hubner, and J. Becker. RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[25] Aws Ismail. *Operating system abstractions of hardware accelerators on field-programmable gate arrays*. Thesis, August 2011.

[26] K. Eguro. SIRC: an extensible reconfigurable computing communication API. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–138, 2010.

[27] M. Jacobsen, Y. Freund, and R. Kastner. RIFFA: a reusable integration framework for FPGA accelerators. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 216–219, May 2012.

[28] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2013.

[29] Xillybus Ltd. Getting started with Xillinux for Zynq-7000 EPP. `http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf`.

[30] Michael K. Papamichael and James C. Hoe. CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 37–46, 2012.

[31] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 25–28, 2011.

[32] Kizheppatt Vipin, Shanker Shreejith, Dulitha Gunasekera, Suhaib Fahmy, Nachiket Kapre, et al. System-level fpga device driver with high-level synthesis support. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 128–135. IEEE, 2013.

[33] Kizheppatt Vipin, Suhaib Fahmy, et al. Dyract: A partial reconfiguration enabled accelerator and test platform. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–7. IEEE, 2014.

[34] Nachiket Kapre, Han Jianglei, Andrew Bean, Pradeep Moorthy, and Siddhartha. Graphmmu: Memory management unit for sparse graph accelerators. In *22nd Reconfigurable Architectures Workshop*, 2015.

[35] John H. Kelm and Steven S. Lumetta. HybridOS: runtime support for reconfigurable accelerators. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 212–221, 2008.

[36] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *Proceedings of the 10th FPGAworld Conference*, page 5. ACM, 2013.

[37] `http://stackoverflow.com/questions/15610570/what-is-the-difference-between-platform-driver-and-normal-device-driver`.

[38] `https://www.kernel.org/doc/Documentation/dmaengine/client.txt`.

[39] `http://forums.xilinx.com/t5/Embedded-Linux/BRAM-DMA-transfer-limitation/m-p/494432/highlight/false#M4496`.

[40] `https://lwn.net/Kernel/LDD3/`.

[41] `http://www.ibm.com/developerworks/library/j-zerocopy/`.

[42] `http://superuser.com/questions/71389/what-is-dev-mem`.

[43] `http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v3_03_a/pg034_axi_cdma.pdf`.

[44] `http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v6_03_a/pg021_axi_dma.pdf`.

[45] `http://www.wiki.xilinx.com/DMA+Drivers+-+Soft+IPs`.

[46] `https://www.kernel.org/doc/Documentation/DMA-API.txt`.

[47] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka8110.html`.