Software for interfacing and management of multiple FPGAs for high performance computing in data centres

Vallavanthara, Amal Jose

Award date:
2018

Awarding institution:
Technische Universität Berlin

Link to publication

Master Thesis

Software for interfacing and management of multiple FPGAs for high performance computing in Data Centres

Student
Amal Jose Vallavanthara
ID: 396462

Supervisor: Prof. Dr.-Ing. R. Orglmeister, TU Berlin
Examiner: Prof. Dr.-Ing. Benno Stabernack, Fraunhofer HHI

Technical University of Berlin, Faculty of Electrical Engineering and Computer Science
Department of Electronics and Medical Signal Processing
Berlin, October 1, 2018

Contents

List of Abbreviations

FPGA	Field Programmable Gate Array
DMA	Direct Memory Access
IRQ	Interrupt Request
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCX	Peripheral Component Interconnect eXtended
MSI	Message Signal Interrupt
PIO	Programmed Input/output
MMIO	Memory-mapped Input/Output
PC	Personal Computer
IOCTL	Input Output Control
SoC	System on Chip
DDRAM	Double Data Rate Random-access memory
RAM	Random-access memory
InfluxDB	Influx Database
SQL	Structured Query Language
IP	Intellectual Property
HPC	High Performance Computing
TFLOP	Tera floating point operations per second
OS	Operating System
DWORD	Double Word
DDR4	Double Data Rate Fourth-generation
SDK	Software Developer's Kit
GPU	Graphics processing unit
TPU	Tensor processing unit
ASIC	Application-Specific Integrated Circuit

Declaration by the candidate

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been listed.

Berlin, October 8, 2018

Signature

Acknowledgement

I thank God almighty for the blessings he had bestowed on me at every stage of my project. I express my heartfelt gratitude to Prof. Dr.-Ing. R. Orglmeister, Technical University Berlin and Prof. Dr.-Ing. Benno Stabernack, Fraunhofer HHI for being my thesis supervisors. I would also like to thank Mr.Steinert Fritjof, Fraunhofer HHI for guiding me through the implementation work. I extend my gratitude to Fraunhofer HHI for employing and trusting me with the project. Last but not the least, I would like to thank my professors and colleagues for the continued support. I thank my family for the continuous support and the belief in me.

Abstract

With the rise of new technologies like blockchain, Artificial Intelligence, Internet Of Things, the demand for high performance computing is at its peak. Optimization in processors are reaching a normalcy. This leads to the need of clustering large number of ASICs, FPGAs, GPUs and TPUs to perform tasks. However, with the increase of cores leads to increase in communication and management overheads. This leads to the need for coherent services and driver management tools for efficient control of the devices and data transfer between the cores. The thesis proposes a driver program which is used for communication between the host PC with multiple Intel FPGAs. The driver will be able to automatically detect and enumerate the FPGAs connected to the host device. Initial steps include sending and receiving data over PIO through PCI express. The overall functioning and optimization schemes of the PCIe will be studied in detail and implemented. Later, DMA access will be implemented for faster data transfer between FPGAs. Choosing the right size of the tokens in DMA transfer is of utmost importance for the overall speed of data transfer and this varies from device to device. Various token sizes for transfer would be analyzed and the best possible case will be implemented. The driver will be capable of management of the different accelerators. The user gets real time information about the parameters of the FPGA. It would also warn the user in case of inappropriate levels of these parameters. This would enable the user to manage, maintain and control the FPGAs connected to the server seamlessly. The SDK will be developed to enable developers to use the functions without the need for understanding the minute framework details. In a nutshell, the work would also provide a software layer that simplifies integration of FPGAs into user applications and environments. It would consist of tools and libraries to discover, enumerate, access, control and manipulate FPGAs. The project would also include test programs to test its usability and to make sure that the management software is implemented in perfection.

1 Introduction

1.1 Application Scenario

FPGA based accelerators are employed in a wide range of industries including communication, aerospace, defense, medical electronics, video and image processing, broadcasting, automotive, security systems, high-performance computing, cryptography, artificial intelligence, blockchain, etc. FPGAs are particularly beneficial compared to ASICs because the cost is much lower especially for low volume productions. The time to market for an application is also tremendously lower. FPGAs are increasingly adopted in enterprises, scalable Data centers and high-performance computing centers in large numbers. FPGAs are adopted as accelerators because complex problems in today's industries cannot be solved by faster CPUs. They require hardware specific acceleration.

These huge enterprises have a heavily armed IT team to manage and maintain the FPGAs. However, many at times, the IT team isn't fully knowledgeable in the hardware level details of the FPGAs. Therefore they need an out of the box solution to manage and control the FPGAs. This brings the need for a comprehensive package which could be installed and used following a simple instruction set. The package should contain pre-developed tools which could be used to obtain FPGA parameters like working conditions, throughput, temperature, etc. A user-friendly GUI showcasing all the above parameters of every FPGA would give the IT personals the ability to manage and maintain all its FPGAs with ease.

The above is from the perspective of IT personals in charge of installation and maintenance of the FPGAs. Another perspective to be considered is from the developer point of view. The developer needs to use the FPGAs seamlessly in their user applications. These developers need not know the hardware level details of the FPGA. They need their user applications to enumerate, access, control and perform operations on the FPGA. They need user applications to read and write data to the FPGA with maximum efficiency. The developer needs to have a test environment to verify if the FPGAs are operating sustainably and synchronously with their application. These would require robust test APIs which can give immediate inspection results. The main purpose of FPGAs is to speed up the process. Therefore parallel programming is an indispensable element of FPGA usage. Therefore parallel programming APIs also need to be developed so that the user application can perform operations on multiple FPGAs simultaneously.

In the particular case, the software is developed for an on-demand cloud-based HPC cluster solution provider which owns huge data centers and rents out its FPGA platform and environment to clients to build test, simulate or run client applications. There would be times when the FPGAs are shared among multiple clients. Therefore it is essential for the different clients to be able to access the FPGAs simultaneously. There are cases where multiple FPGAs need to be accessed by the same client simultaneously. Therefore serial and parallel programming should be possible hand in hand. It would be an added selling point for the solution provider if the clients could view the performance of the FPGAs from anywhere around the world through a user-friendly interactive GUI.

In many cases, the performance of the driver needs to be varied. For instance, in cases of usage within data centers, a huge amount of resources such as memory and CPU power are available. Therefore, the driver would work at maximum efficiency in terms of latency. However, if the FPGAs are to be used for development in a resource constrained server, the available memory, power and cycles are limited. Therefore the driver should trade-off overall speed to balance out all the other requirements. This arises the need for performance altering of the system driver. In many enterprises, altering the performance of FPGAs in real time is also a necessity. For instance, during the daytime, services which are customer oriented should be prioritized. But during nighttime, when the customers are not so active, operations and services within the enterprise could be provided more resources. Therefore the driver software should be able to modify its resource usage in real time without affecting the sustainability of the platform.

1.2 Definition of Goals of the work

Clear cut goals of the project were pre-defined so as to ensure speedy and efficient development:

- Build a Character Device Driver to control and manage the FPGAs connected through PCIe to the host server.
- Huge Data centres would have hundreds of FPGAs running in parallel. The driver should detect and enumerate the FPGAs for management.
- The driver should be able to perform PIO read and write to each of the 16 sockets of the FPGAs. The number of sockets might vary depending on the FPGAs.
- The driver should be able to perform DMA read/write operations from the user space of the host server to the FPGA. This DMA transfer should be optimized in terms of throughput, power consumption, latency, memory requirements, etc.
- The driver should be able to switch between polling and MSI interrupts to determine completion of DMA transfers. MSI interrupts are efficient, but more complicated and less versatile. Therefore depending on the platform the user should be able to switch between the two.

- User space APIs to control and manage the FPGAs are to be developed. These APIs gives the users the ability to control and manage the FPGAs without the need to know the hardware level details. This enable speedy development of softwares and applications.
- User friendly APIs and informative Documentation allows easy integration of the FPGAs and its APIs into user applications.
- The driver could be used in different kinds of platforms and environments. Therefore depending on the performance requirement, the performance of the FPGA need to be altered.
- There are instances where several FPGAs need to be controlled from the same application simultaneously. Separate APIs need to built be for this purpose.
- A test environment needs to be built which would test the sustainability and portability of the FPGA, the driver and platform. Test APIs are required to verify if all the above requirements of the software are fulfilled all the time.
- For efficient control and management of the FPGAs, a user friendly GUI is to be developed which would showcase FPGA parameters like temperature, performance level, read and write throughput from anywhere around the world.

1.3 State of the Art

The section analyses the technological analysis and literature work done by experts in the field. [KB12] shows a PCIe device driver using Xylinx FPGA used in IceCube Neutrino Observatory(South Pole). It has a performance of upto 784MB/s for a 4 lane generation PCIe bus. The [KMB14] also shows that the performance can be tremendously increased by using a motherboard with a better chipset. The size of the buffer would also increase the data rate. In 2014, [RCC+14] shows PCIe core adopted a new strategy wherein the DMA descriptor list is stored inside the FPGA rather than the central memory. With this approach a peak data transfer rate through the PCIe of upto 3.4GB/s was achieved. [ZLBAM15] uses IOCTL functions to control the DMA read and write operations from the user space.

[CPCAKMN18] writes about the PCIe DMA design for an FPGA developed for the detector system at CERN LHC designed to study the properties of Quark Gluon Plasma. It is the interface between the online-offline computing system and Common Readout Unit(CRU). The design involves loop back flow of data for high performance data transfer. The length, source and destination address are programmed into the descriptor table. The descriptor table is then transported into the FPGA. Once the data transfer according to the descriptor is complete, it updates the status register with the value 1. The paper mentions that a major bottleneck for the implementation and increased latency was the register configuration for source and destination addresses. This created dead time between two consecutive DMA data transfers. They reduced the dead time by looping over descriptor tables. This ensured pipelining of descriptors. A series of tests were conducted to choose the appropriate number of descriptors for maximum performance. Finally 128 descriptors were chosen and they were able to achieve upto 6GB/s throughput in Gen3 8 lane mode. The design needs to make sure some additional processes are performed to make sure that dead time for DMA transfers is limited. [CPCAKMN18] states that the polling to check DMA status affects the DMA performance considerably. It also states that future work is to consider developing the same project on PCIe40 card.

[PDL15] implements PCIe readout with a throughput of upto 12GB/s using 16 lane PCIe. The driver is developed for the LHCb experiment at CERN. The current hardware clocks at 1.1 MHz and they need an upgrade to 40 Mhz. After careful consideration, they chose the PCIe Gen3 technology. Performance analysis on i7-3820 CPU and i7-4770 CPU, shows a variation of upto 10 percent. This shows the performance of the DMA operation is highly dependent on the host platform as well.

2 Theoretical background

2.1 PCI express and Nallatech Board

2.1.1 PCI Express

PCIe was designed to add more features and performance to the older PCI [RB05]. PCIe was designed such that it would be software compatible with PCI. This was done to make sure that the older systems would be able to detect and configure the new PCIe without the PCI express features. Therefore PCIe was designed with the same usage model and software compatibility. However, they do come with differences, for instance, PCIe is designed on serial bus technology whereas PCI was based on parallel bus [Abr15]. This reduced the number of lines connecting the PCIe endpoint to the host PC thereby reducing the complexity and cost. This increased the transaction frequency thereby increasing the overall read and write throughput. Generation 3 of PCIe supports upto 8 GBps of transaction rate theoretically. It has scalability where by the number of lanes used per device can be varied. The data-sheet provides the specification for usage from 1 lane all the way upto 32 lanes. The lane width and the speeds are negotiated between the host complex and the endpoint device during the enumeration process. Either the end point card or the root complex can decide on the maximum lanes and throughput to be used. It is to be noted that both the PCIe and PCI are only software compatible. Therefore a PCIe device cannot be inserted into a PCI slot or vice versa.

Root complex is basically the host controller present on the server and provides slots on which other PCIe cards can be connected. PCIe has its own memory address space which can be either 32 or 64 bit. This address space is visible only to the PCIe devices such as root complex, PCIe endpoints, switches and bridges. Switches are used to connect multiple PCIe devices to the root complex. These switches can be used when the board doesn't contain sufficient slots for all the endpoint devices.

2.1.1.1 Structure

The root complex connects the external PCI and PCIe devices to the CPU [Abr15]. The CPU initially accesses the root complex to configure the root complex IP and then finally the external devices connected through the root complex [RB05]. Once the endpoint devices are configured, the CPU uses the root complex for all further transactions with the endpoint.

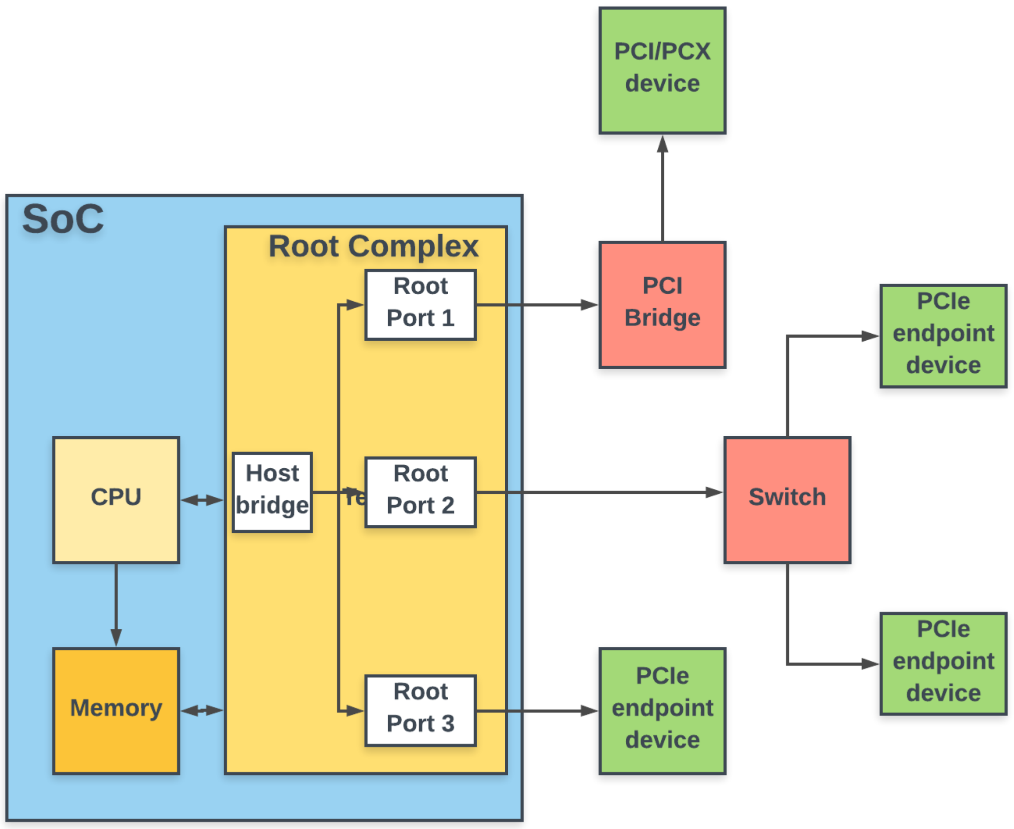

Figure 2.1: PCIe structure diagram

The root complex can send interrupts to the CPU for any event completion of the root complex or of any of the endpoint devices. The root complex has direct access to the PC memory. This capability is employed to perform high-performance DMA transfers between the endpoint devices and the host PC. To enable DMA transfer, the root complex must first provide the 'Bus Master' status to the endpoint device. Further, it should send addresses in the host PC's memory to which DMA transfer should be performed by the endpoint device. Devices are to be connected to the root complex through the root ports. Figure 2.1 shows the particular root complex has three root ports embedded into it. The first root port is connected to the PCI device through the bridge. PCI-X (Peripheral Component Interconnect eXtended) devices can also be connected using PCX bridges. Multiple PCIe devices are connected to the 2nd root port using a PCIe switch. A PCIe device is connected directly to the third root port.

PCIe uses point to point topology. In the mechanism, a single point to point bus would connect two devices. The root complex contains a host bridge which connects the root ports to the CPU. The root ports are all simple Virtual PCI-PCI bridges. A root complex may have multiple Virtual PCI-PCI bridge. The host bridge is connected to the Virtual PCI-PCI bridge through Bus 0. Each of this Virtual PCI-PCI bridge spawns separate buses through which other PCI devices can be connected. A switch inturn has multiple Virtual PCI-PCI bridges. The root complex is connected to the Switch through the Virtual PCI-PCI bridge which in turn spawns out a new bus. The multiple endpoint devices are connected to the switch through new Virtual PCI-PCI bridges which in turn spawns out a new bus for each of the devices. Each of the buses would be given a bus number or an ID during the enumeration process by the host PC. These given buses numbers will be used henceforth for routing the data. Each bridge or switch contain information about 3 buses [KMB14]:

- Primary Bus number: The bus number through which the particular device is connected to the root complex.
- Secondary Bus Number: The lowest bus number which is greater than the primary bus number.
- Subordinate Bus number: The highest Bus number which is connected to the switch or bridge.

Any transaction to a bus number equal to or greater than Secondary Bus Number and less than or equal to Subordinate Bus number will be routed through the switch/bridge [Zha10]. PCIe uses lesser number of lanes compared to the PCI because of serial transactions. Each lane consists of 2 pairs of RX and TX lines. PCIe, unlike PCI, has no separate IRQ lines.

2.1.1.2 Address Space

The root complex can access the PCIe configuration space using the PCIe address space [Abr15]. The PCIe configuration space provides the server all the required information about the device. This information is used to enumerate the device and to allocate sufficient resources to the device by the server. PCIe address isn't a physical entity and is completely virtual. It is a list of information used in the transaction layer packet to identify the destination and source of the packet. The root complex comprises of root registers which store the IP of the devices. It registers information like the clock frequency, the vendor ID, device ID, lane widths, transaction speeds, address translation unit, etc. The address translational unit is used to translate the CPU address space to the PCI address space and vice versa. The root complex also has the configurable address space. It is used by the CPU to access the PCIe address space. Every root complex contains the following 4 address spaces associated with them according to PCI specification:

- Configuration Space
- IO space
- Message Space
- Memory Space

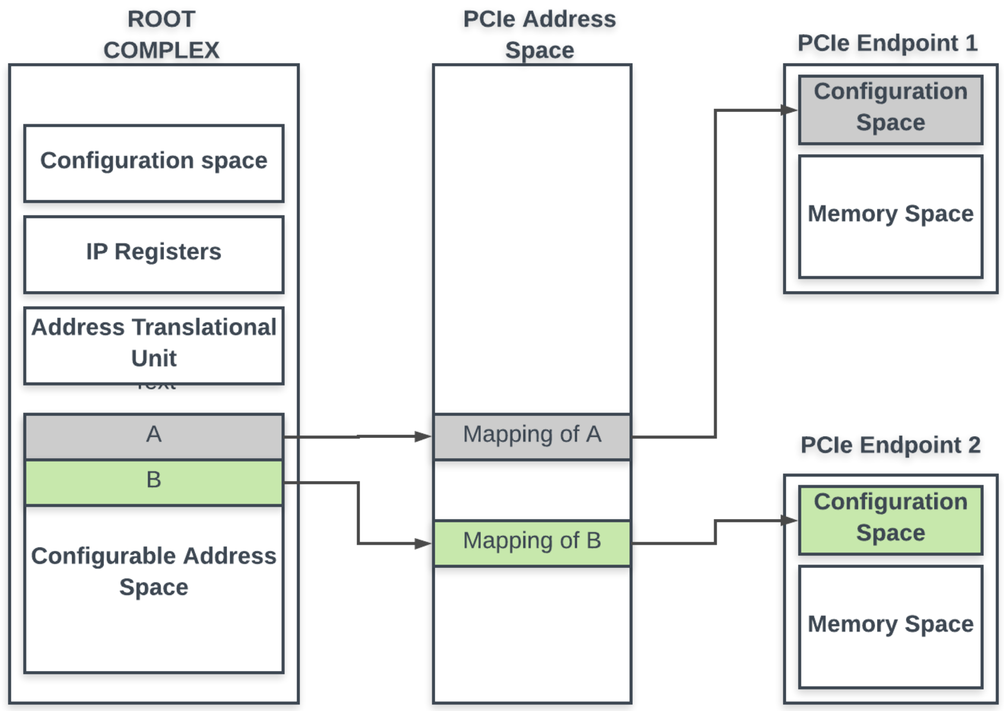

Figure 2.2: PCIe Address Space

All the above address spaces except memory space have physical addresses associated with them. The configuration space contains all the information required by the root complex about the device. This includes device ID, vendor ID, MSI capability, device class and other various capabilities of the device. It has various registers to configure the capabilities of the device. For example, changing the configuration values can put the device to a lower power state or disable interrupts. These kinds of additional capabilities were added to PCIe by including a bigger configuration space. The size was increased from 256 Bytes (in PCI) all the way up to 4KB. The first 64 bytes is called standardized headers and are of 2 types-type 0 and type 1. Type 1 is used by root ports, switched and bridges. Type 0 is used by the end point devices and holds all the information regarding its configuration. A PCIe Address Space is created as shown in Figure 2.2. it maps the Configuration space of the endpoint devices to the configuration space of the root complex memory. This mapping is virtual and there is no physical PCIe addresss space as such.

2.1.2 Compute Acceleration Card - Nallatech 510T [1718]

It is an accelerator card for high-performance applications in Data Centers. The board is specifically designed to house 2 FPGAs (Intel Arria 10) to deliver maximum per watt performance. It uses 16 lanes Gen 3 PCIe cards. The 16 lanes are split up among both the FPGAs. Its characteristics are as follows :

- Performance upto 3 TFLOPs
- 75 GBps Peak DDR4 Memory bandwidth
- Upto 150 GBps Aggregate memory bandwidth

It has already been employed in the following markets:

- HPC
- Datacenter
- Compute, Network and storage
- Communications
- Industrial, Broadcast
- Automotive
- Medical

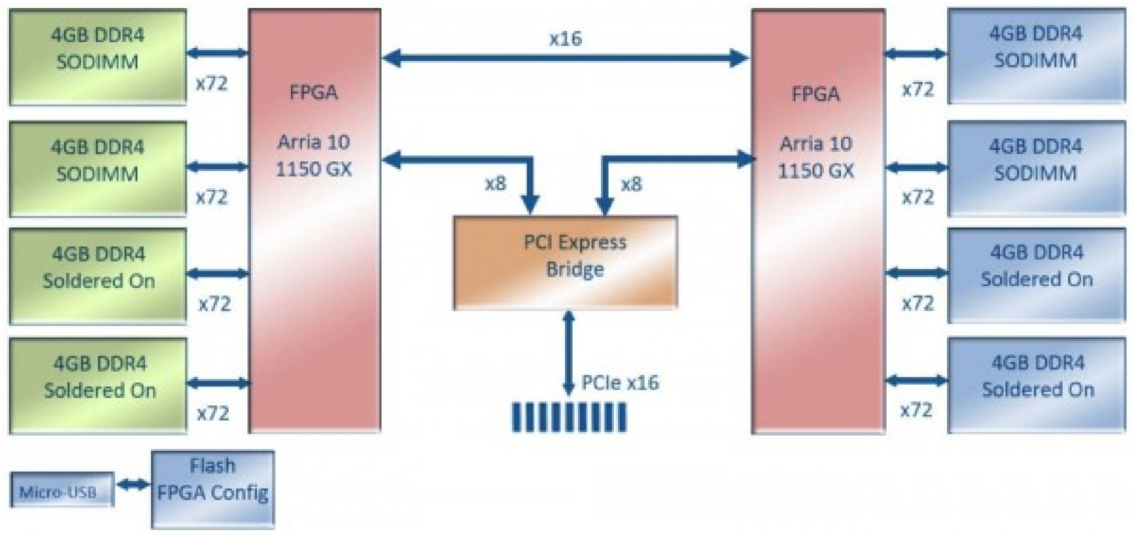

Figure 2.3: Nallatech Board

The actual performance of data transfer between FPGA and server is highly dependent on the host server's hardware specifications and operating system. It contains 8 banks of DDR4 SDRAM. Each of the bank is of 4GB. Therefore, the system houses a total of 32GB memory with a transfer rate of 2133 MT/s. The card is designed to deliver upto 225W power consumption. But the actual power consumption is highly dependent on the application running on it. The power is derived from 12V PCIe slot and one 8 pin AUX connector. It has inbuilt sensors to detect voltage and temperature variations. The board is designed so that the developer need not know the hardware level details and can instead use the top layer abstraction for speedy software development.

2.2 Linux Kernel - Memory allocation and parallel programming

The kernel is the core of the PC that acts as an inter-connecting bridge which connects the user programs/applications to the underlying hardware. In short, kernel is where the operating system of the device operates. The kernel performs the management operations at the hardware level including data processing, memory and task management. Since a number of applications need to use the hardware at the same time, proper synchronization and control of the kernel are of paramount importance. Clear steps should be taken so that no application can take control or hack into the kernel and control the rest of the system. The segregation is also done to make sure that no bugs in any userspace application affect the rest of the system. Thus, the system memory is divided into two regions:

1. Kernel Memory
2. User Space Memory

2.2.1 Kernel Space memory

The operating system runs in the kernel of the system. Usually, in x86 architectures, there are 4 rings of operation. Ring 0 - Kernel Mode, Ring 1 and 2 - usually used by Device drivers, Ring 3 - User Mode. Ring 1 and 2 have "higher" privileged mode of access in a way that they can use kernel memory space and hypervisor pages, but they can't use a privileged instruction. So this is the ideal place for drivers. The initial section of the kernel memory is contiguously mapped and is called the lowmem. This is done to help in easy memory mapping for transfer of data. When a memory is required by an application, it is not continuously allocated in the kernel memory. It is allocated based on the flags the user specifies and the available free space. This is performed with the mechanism of virtual addressing. Virtual addressing gives the user application the notion that all the allocated memory is continuous even though in the actual physical memory the allocation is disjointed and separated. The virtual memory gives the user the abiloity to use some parts of its hard drive as if it were its RAM.

The user application is given continuous 'virtual' addresses by the Memory Management Unit. The page table maps the virtual address to the actual physical address. The memory is divided into fixed small sized blocks called pages. Memory should be allocated only in multiples of page sizes for DMA transfer. The page size in the x86 architecture used for the development of driver is 4096 Bytes. In x86 - 64 architectures, the virtual memory has address width of 64 bits. However, only the least significant 48 bits are valid and provide the entire needed information about the actual location of the physical memory. All the bits from the 47th bit to the 63rd bit are copies of the exact 47th bit. Therefore the addresses span from 0x00 to 0x00007FFFFFFFFFFF, and from 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. The userspace usually occupies the lower part of the memory. The kernel space memory occupies from the upper part of the memory from 0xFFFFFFFFFFFFFFFF and expands downwards [DSM18].

This is also a major concern during memory mapping for data transfer to and from 32-bit external devices. The memory mapped from the 32-bit devices may not be able to access the kernel memory because it can't tap 64-bit addresses from 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. Thus naive memory mapping wouldn't work. Therefore in such special cases, care must be taken to make sure that the memory allocated in kernel space is in the lower part from 0x00 to 0x00007FFFFFFFFFFF. This can be done in 2 ways:

1. Set up the driver configuration such that the driver is loaded during boot time. This gives a higher rate of successful allocation as the memory is usually free during boot time and other applications may not have already procured the available memory. This, however, is not suitable for a driver which needs to be modular and portable.
2. Use _GFP_DMA while allocating using kmalloc. The memory of the kernel space is divided into 3 zones: normal zone, DMA capable zone and high memory zone. Usually kmalloc allocates memory in the normal zone. But setting flags would make sure that the memory is allocated in either of the other two zones. These zones are platform dependent. In x86 architectures, DMA zone lies in the first 16MB of the RAM. This is also dangerous because the allocation fails in case the first 16MB of RAM is already filled.

```
#include<linux/slab.h>
void *kmalloc(size_t size, int flags);
```

Kmalloc allocation can be made contiguous in physical memory by using the right flags [5018]. In x86 architecture, the maximum contiguous memory that can be allocated is 4 MB. Any allocation more 4 MB needs to be allocated separately. In the particular implementation which uses more memory, the memories are allocated separately of 1MB size each. The argument of kmalloc 'size_t size' is the length of the buffer that needs to be allocated and the second argument 'int flags' is the flags that need to be considered during allocation. One can also pass multiple flags using the OR operation. GFP_KERNEL is the flag used for allocation. This is used particularly in cases where the allocation is not needed to be atomic. This is because, in case the memory allocation fails, kmalloc puts the process to sleep until sufficient memory is freed to allocate successfully.

2.2.2 User Space Memory

Userspace memory also popularly called userland is where all the user applications are implemented. They have no access to the kernel functionalities unless used in privileged mode. All the applications written in this space need to be transferred to the kernel space before they can be used by device drivers for passing the data to the external devices through the PCI express. This can be achieved by using the copy_to_user and copy_from_user functions available in the library include <asm/uaccess.h> [5018].

```
int copy_to_user(void *dst, const void *src, unsigned int size);
```

The copy_to_user function copies 'size' amount of bytes from the kernel memory space pointed by src pointer to the user space memory location pointed by dst* [Mad17]. It initially checks if the source and destination addresses are accessible. If they are accessible then, it moves on to check if the memory location is pinned to memory because the virtual address page table could change at any time.

```
int copy_from_user(void *dst, const void *src, unsigned int size);
```

The copy_from_user function copies 'size' number of bytes from the user space to the kernel space memory pointed by the src and dst pointers [Mad17]. The kernel page fault handler treats this as an inaccessible address rather than a kernel code bug if the memory is not accessible by the function. This is an added ability/feature of the function. The user space applications can communicate with the kernel through system calls. In addition, ioctl(input-output control) function can be used to control the modules in the kernel.

2.2.3 Parallel programming

For simultaneous access of multiple FPGAs, parallel programming is an indispensable tool. This can be achieved in the systems by using multi threading. A thread is an independent stream of instructions that can act independently from the main function [BB18]. On running a developed program, the process is created by the operating system.A process contains an overhead with the following parameters:

1. Process ID, process group ID, user ID, and group ID
2. Environment
3. Working directory.
4. Program instructions
5. Registers
6. Stack
7. Heap
8. File descriptors
9. Signal actions

10. Shared libraries
11. Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

A thread uses some of the above parameters owned by the main function. They run as independent entities and duplicate only some of the above basic resources (Registers, Stack pointers, Scheduling properties, etc). Pthreads is a standard specified by the IEEE POSIX 1003.1c standard (1995) for multi threading applications. Creating and managing threads can be performed by much lower resources and overhead as compared to a process [BN11]. Pthreads enable efficient data transfer between different entities. Any number of threads can be created using the following command:

```
pthread_create(&tid, NULL, start_fn, arg);
```

It is common that the different threads try to access the same registers or resources. If multiple threads access access and change the memory simultanoeusly, it could cause inconcourrency in the memory. Therefore the idea of mutex is introduced. This makes sure that once a thread accesses and locks a resource, it cannot be altered by any other thread. This locking and unlocking can be performed by using the following command [Ber96]:

```
pthread_mutex_lock(m)
... ...
pthread_mutex_unlock(m)
```

2.3 DMA Memory mapping

There are basically two types of mapping for DMA read/write:

1. Streaming DMA mapping

   This kind of mapping is done when the mapping is done right before a read or write is performed and then once the task is complete, it frees the memory. This provides the capability to optimize immensely for an application-specific task. However, the process of mapping and unmapping for every read and write creates too much overhead and decreases the overall efficiency.

2. Coherent DMA Mapping

   In this mechanism, the mapping is performed on detection and enumeration of the device initially. Once the mapping is performed, it remains until the memory is freed during unloading of the module. This is specifically done when both the device and the PC needs to see changes made in the memory in real time. The memory needs to be updated with the need for separate software flushing. Most importantly, memory access should be possible to access by both the device and the PC in parallel. Therefore the mechanism is also called synchronous mapping.

All the above arguments prove that Coherent DMA mapping is the best option for the particular implementation. The first step to mapping is to check if the device is capable of reading any memory location in the kernel. 32 bit external devices usually wouldn't be able to access locations in the upper part of the kernel memory from 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. The PCI device capability to read this section is highly device dependent even for 64-bit external devices. This can be checked by using pci_dma_supported() function available in the linux/pci.h library. If the device supports only the bottom 32 bit addressing then the kernel can be informed about such special cases by using the dma_set_mask() function. Once this is done, the actual mapping is done using pci_alloc_consistent() [5018]:

```
void *pci_alloc_consistent(struct pci_dev *pdev, size_t size,
                                dma_addr_t *dma_handle);
```

The function allocates a buffer of length 'size' for the device 'pdev' for the bus address *dma_handle and returns a virtual address that points to the bus address and can be used by the driver for all further operations. The FPGA memory is not cache coherent to the PC because the onboard DDR-RAM could change independently. The memory should be deallocated before the device driver is unloaded. A maximum of 4096 bytes of data can be contiguously allocated by the PC. In case more memory needs to be DMA mapped, they should be broken down and split into smaller non-contiguous memory blocks as is implemented in later sections. The DMA controller used by the FPGA uses scatter gather mechanism of DMA transfer. By scatter gather mechanism, it enables data from different locations to be transferred in bursts to locations that are separated from each other. Thus the data is being gathered from different locations in the source memory and then scattered in different locations in the destination device memory.

3 Specification

3.1 Driver Development

A device driver controls, manages, directs and monitor an external device connected to the host PC. An external device like a mouse, FPGA, keyboard can be controlled by a software or another hardware which in turn is controlled by a software. The FPGA in this implementation is connected to the host PC through the PCIe bus. A device driver is broadly divided into 2 parts :

1. OS Specific
2. Device Specific

The OS specific part basically deals with how the kernel deals with the external device. It is highly dependent on the operating system. Thus, a Linux, MacOS and windows driver vary highly. The device specific part of the driver is the same irrespective of the OS platform the driver is hosted on. The part needs to be programmed based on the data sheet provided by the vendor. The data sheet includes specifics like the programming, operations, registers, performance, etc.

The device driver in Linux provides an interface for the user in the user space to the kernel space through system calls [XL17]. This system calls is the methodology in which users can interact with the kernel to control, modify, operate a hardware related service, creation or operation of new processes or even basic scheduling of processes. In Linux systems, the drivers can be broadly classified into three:

1. Packet-oriented
2. Block-oriented
3. Byte oriented or character devices

The particular implementation uses the character vertical because the application demands byte-oriented accessibility. Linux character drivers have an advantage that they can be loaded and unloaded on the fly. In the case of windows, the system needs a reboot to enable and disable the device driver. The kernel architecture of Linux is designed such that these dynamically loadable drivers are called modules and built into individual files. These files have a .ko extension and are called kernel objects. To dynamically load or unload a module, the user can use the following commands

```
insmod <kernel object> - loads a driver
rmmod <module> - unloads a driver
lsmod - lists all current drivers
modprobe <module> - loads the module and all its dependencies
```

These modules are located by design in the /lib/modules/<kernel_version>/kernel in the root directory.

The driver does not contain a main() function. It uses the headers files from the source code of the kernel and not from the standard '/usr/include' as it is linked/loaded to the kernel [5018]. The kernel used for the particular implementation is 3.10.0-862.el7.x86_64. The module's constructor is invoked on loading the file using *insmod.* The destructor is called on unloading the driver using *rmmod.* These two functions are named by module_init() and module_exit() in the kernel headers [Lae12]. Note that to build the file one needs to have the kernel headers installed on the host PC. It is usually installed at /usr/src/linux and should be verified before building the driver. The developer can also install the source code using the command line [5018]:

```
Linux -  yum install kernel-devel
Mandriva - urpmi kernel-source
Ubuntu - apt-get install Linux-source
```

The driver uses *printk* statements to debug and print information to the user. The syslog daemon picks the messages from the log buffer and redirects the messages to the devices based on the configuration file /etc/syslog.conf. The developer has the option to print the statements only during debugging rather than the bombing the kernel's log throughout. These can be controlled by setting the log levels for each *printk* statement. The various *printk* log levels are given by the following table.

The developer has the option to set each *printk* statement to different levels. The developer can choose to view only a certain level or higher log messages during debugging. The log level helps the kernel determine if the message is of utmost importance or not even necessary at all and can decide if it should be shown at all. To change the log level to be shown, the user can simply write into the file /proc/sys/kernel/printk. Reading this file gives information regarding the log level chosen by the developer for the particular console and the default log level.

Macros __init and __exit is used at the beginning of the functions to show that it is to be built along with the kernel. Thus the system loads the driver on bootup and unloads the driver during system shutdown. These functions are executed once on bootup and shutdown. It is not relevant in case the driver is to be dynamically loaded and unloaded. All functions with the __init keyword are saved in the init section of the kernel. This optimizes the system as once it is loaded during boot time, the function is removed from

Name	String	Meaning
KERN_EMERG	"0"	Emergency messages, the system is about to crash or is unstable
KERN_ALERT	"1"	Something bad happened and action must be taken immediately
KERN_CRIT	"2"	A critical condition occurred like a serious hardware/-software failure
KERN_ERR	"3"	An error condition, often used by drivers to indicate difficulties with the hardware
KERN_WARNING	"4"	A warning, meaning nothing serious by itself but might indicate problems
KERN_NOTICE	"5"	Nothing serious, but notably nevertheless. Often used to report security events.
KERN_INFO	"6"	Informational message e.g. startup information at driver initialization
KERN_DEBUG	"7"	Debug messages
KERN_DEFAULT	"d"	The default kernel loglevel

Table 3.1: printk Message logging Levels

the RAM. This optimizes the PC because if the host PC is shutting down it doesn't need to clear the RAM anyway. The kernel can decide if the exit section needs to be invoked at all based on the situation, thus optimizing the host PC. A user can communicate with the driver of any device only through its character device file linked to it using the Virtual File System [JC05]. When the user performs any operations on the device file, it is translated by the VFS to its corresponding functions in the kernel space. These functions perform the required actions in the low hardware level.

3.1.1 Connecting User Application to FPGA

A complete connection from the user to the endpoint device actually involves the synchronized connection of the following four entities:

- Application
- Device File
- Device Driver
- FPGA

Each of these entities can exist independently but to truly connect from the user to the FPGA, each of the entities should be linked to each other. The driver is linked to the FPGA by low-level device specific operations. The driver is linked to the device file by appropriate registration. The application connects to the device file by using open system call functionality from the userspace. The application can connect to the device file using the file name. The file name in the implementation is 'intelfpgadev<fpga number>' By

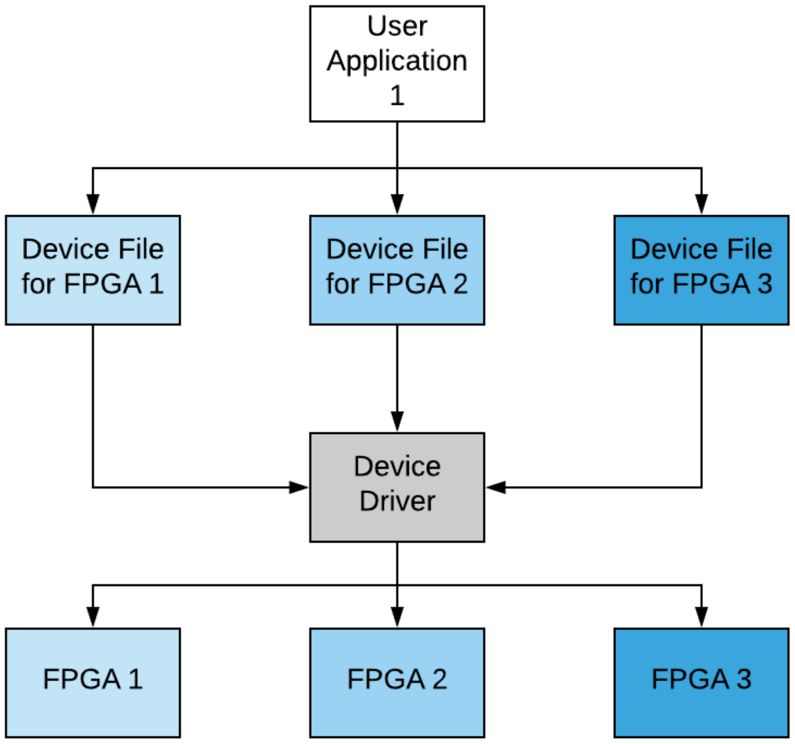

Figure 3.1: General Working Scheme of a Driver

using the required <fpga number> the application can connect to the device file of any of the FPGAs. The connection between the device file and device driver uses a combination of numbers called the major number and minor number to establish the link. It is a pair <major, minor> defined as dev_t in linux/types.h. The driver can be connected to the device file by the following steps [JC05]:

1. Register the major and minor number of the device file.
   The following function dynamically allocates a free major number to the device file and sets cnt as the minor number [5018].

   ```
   int alloc_chrdev_region(dev_t *first, unsigned int firstminor,
                                      unsigned int cnt, char *name);
   ```

2. Link the device file operations to the driver functions.
   Device files were created automatically by the kernel itself in earlier versions of Linux. However, from kernel version 2.4, the kernel only populates the device information and device class into the /sys window. The udev uses this information to create the device file [1218]. Udev can then be configured to modify the device file names, permission and types. The driver initially only needs to set in the right information into the /sys window. The device class can be created by using the following function [5018]:

```
struct class *fpgaclass = class_create(THIS_MODULE, "FPGA_class_name");
```

Further, the device major and minor number can be linked by using the function:

```
device_create(fpgaclass, NULL, <device number>, NULL, "<device name>");
```

On unloading the driver initially the device should be destroyed and then the class should be deleted. The following functions should be invoked in the same order:

```
device_destroy(fpgaclass, <device number>);
class_destroy(fpgaclass);
```

The device file is like any other Linux file and hence any operation applicable on an ordinary file is possible with device files. The only difference is that for device files, there is a VFS layer underneath which converts the operations into functions in the kernel level. The VFS should be informed about the link between the file and driver. This basically consists of two steps[1218]: Populate the file operations structure (struct file_operatons) with the desired operations to be performed by the driver (.open, .close, .read, .write, .probe) and then initialize the character device structure using cdev_init(). Finally pass the initialized structure to the VFS using the function cdev_add() from the header <linux/cdev.h> [JC05].

3.1.2 Read and write the Device File

By using the above functions, the system creates the following device files in /dev

```
/intelfpgadev1
/intelfpgadev2
```

Now on performing a read operation using the .read from the device file, the VFS in the kernel decodes the major and minor number and directs the message to the appropriate driver function defined by the structure file_operations. It performs the read operation that has to be hardcoded by the developer in the driver. It finally returns the number of bytes that is read. The driver reads the data from the external device and then writes them into the specified buffer in the user space. The write operation performs the opposite. The user specifies the number of bytes to be written from the user space into the external device. The system actually reads the data from the user space and then writes them into the external space [Buc04].

3.1.3 Input-output Control

Input/Output Control (ioctl) is a system call available in almost all driver categories [1218]. The advantage is that it can be used for all control operations of the external device. In addition, it can be used for changing the parameters or operations of the driver by itself. This is achieved by its arguments: command and argument. The command is the number that represents the action to be performed. The arguments are the list of values that need to be passed from the user space to kernel space or vice versa to perform the operations. The ioctl() function implements a switch() such that based on the command number, it can choose what operations to perform. The following is the function definition [5018]:

```
long ioctl(struct file *f, unsigned int cmd, unsigned long arg);

Where:
struct file *f  is the pointer to the device file
int cmd - command number for the operation to be performed
unsigned long arg - structure containing all variables to be passed
                            from user to kernel space or vice versa
```

It is initialized by filling the structure file_operations with the appropriate function pointer .ioctl just like .read, .write, .open, described earlier. The function is invoked from the user space just like pread, pwrite, open, close functions. The developer should use a number that is unique throughout the system to issue an ioctl command. Else, it could cause errors such as issuing the right command to the wrong device. To create unique command codes, the bitfields of the command number has been split up. It uses 4-bit fields which have the following meaning [1218]:

```
[bits 31:30] Direction: The field depicts the direction in which data
             is transferred.
_IOR - Data is transferred from kernel space to user space.
_IOW - Data is transferred from user space to kernel space.
_IO  - Data is transferred in both directions.

[bits 29:16] Size: The size of the argument that is passed.

[bits 15:8] Type: This is the 8-bit magic number and should be used throughout.
            It is 8 bit wide.

[bits 7:0] Original command number - defined as per our requirement.
```

Another issue that arises with the ioctl function is the second argument. The second argument passes a pointer to a structure with several variables. It is the driver's duty to make sure that the pointer in the user space is accessible by the driver in the kernel space. This can be checked by using the access_ok() function. It checks that the pointer is located in a region that is accessible by the driver. It also ensures that the pointer is not present in the kernel space. In the implementation ioctl is used for the following purposes by including the command number in system calls :

FPGARDGETTRANSFERRATE
To obtain the transfer rate of the last DMA read in GB/s

FPGAWRGETTRANSFERRATE
To obtain the transfer rate of the last DMA write in GB/s

FPGAMSIINT
To manually enable MSI interrupts to determine read or write completion

FPGAPOLLINT
To manually enable polling to determine read or write completion

FPGAWRITEDMA
To write data from user space to FPGA through ioctl function

FPGAREADDMA
To read data from FPGA to user space through ioctl function

FPGAPERFORMANCE
To control the performance level of the FPGA driver. The driver has 3 levels of operation based on memory and speed constraints.

FPGATEMPERATURE
To read the temperature of the FPGA through ioctl function

FPGAGETPERFLEVEL
To obtain the performance level that the FPGA driver is currently working on

FPGAGETNUMOFDEVICES
To obtain the number of FPGAs that the driver has successfully enumerated and connected

3.2 DMA transfer

3.2.1 Design of DMA transfer

Implementation of DMA transfer between the host PC and the FPGA is the precarious and most complex section of the implementation. The user has all its data in the user space. However, there is a separation between the user space and kernel Space. A possibility to solve the issue is by mapping the user space to the kernel space from which data is transferred [XL17]. However, this mapping of DMA for character device files causes a lot of complications as shown in section 3.2.1.1. Therefore the data needs to be copied from the user space to the kernel space and then to the FPGA. This causes an increase in latency in data transfer and decreases the performance of the whole system because of the additional copy operation. Efficient algorithms need to be implemented to make sure that the data transfer from the user space to kernel space and from the kernel space to the FPGAs take place parallely without causing much utilization of the PC and time overheads. From a software engineering point of view, the efficiency can be increased by the formula [RG95]:

$$Tn = [f + (1 - f)/N] * T1$$

-Where f is the task that cannot be parallelized. $1 - f$ is the section that can be parallelized by$N$ different threads/processors. Then Tn is the efficient timing derived from parallisation by N threads and $T1$ is the naive timing.

However, the optimization task has to keep in mind the available resources as well. In huge data centers with plenty of available resources in terms of memory and CPU power, timing efficiency is the only parameter to be considered and optimized. But in cases where the FPGA is to be used on the go, optimization should be done considering other parameters like memory usage, power optimization, etc.

DMA transfer has to be done in two phases:

1. Data needs to be transferred between user space and kernel space
2. Data needs to be transferred between kernel space and the FPGA

3.2.1.1 User Space to kernel space

The first question that arises is can't the user space and kernel space be memory mapped. It is not the best option for character devices because of the following reasons.

1. In kernel mode, the code has complete and unrestricted access to the memory and underlying hardware. This is a serious threat as the it would create vulnerabilities

to malicious activities. Hackers could use the loophole to disrupt the security of the device.

2. Crashes in Kernel mode are catastrophic, they would crash the whole PC and in worst case affect the operating system is such a way that there is no comeback. So it wouldn't be a good idea to let a device driver to device the fate of the entire PC where many applications run in parallel.
3. Kernel Mode code has higher performance. The only extra time consumption is the time taken to transfer data from user space to kernel space. If this latency is overcome, using driver in kernel space is beneficial. The following section elaborates on how to reduce this time and increase performance.

3.2.1.2 Kernel Space to FPGA

The ideology for DMA mapping is that an external device will be able to perform and control the data transfer without the CPU usage of the server [Buc04]. By letting the external device control the DMA transfer, the PC doesn't lose clock cycles and can be revert to perform more indispensable tasks. The DMA transfer is performed by the DMA controller in the FPGA using 128 descriptors (Detailed explanation in section 3.2.2).

The driver is implemented by 3 different mechanisms. The DMA read and write command can be performed from user space by using the APIs developed. The 2 APIs used are:

```
void dmaread(ssize_t file, char  *buf, u_int64_t len, u_int64_t off)
void dmawrite(ssize_t file,char  *buf, u_int64_t len, u_int64_t off)
```

Dmaread API transfers data from the host PC user space to the FPGA. Dmawrite API transfers data from the FPGA to the host PC user space. The user has to pass the following arguments while invoking the function in case of DMA read:

- Driver file opened using opendriver function (the number that is used to reference the FPGA)
- The pointer to the data that is to be transmitted.
- The size of data to be transmitted
- The offset address location in the FPGA to which data needs to be transmitted.

In the present driver, the read is done through driver built-in function .read. Additional capability is also provided for the driver to implement using input-output control functions (ioctl). In the kernel driver, the function copy_from_user is used to transmit data from the user space to the kernel space [Lin12].

The next step in DMA transfer is to calculate the number of descriptors required to transmit the data. Each descriptor can send a maximum of 1MB of data. There are only 128 descriptors in total. So in case, the user needs to transmit 8.5MB of data, the driver needs to deploy 9 descriptors. The first 8 descriptors transfer 1MB each. The final descriptor transmits the last 0.5MB of data. In cases where the user needs to transmit more than 128MB of data, complex algorithms are developed such that the data is transmitted in blocks of 128MB each. Each block consists of 128 descriptors which transmit 1MB of data each. Algorithms are also developed such that the offset address from which data needs to be read and the destination address to which the data needs to written is calculated for each descriptor within each block.

```
Total_Number_of_Blocks  = Total_Data_Size / 128
Total_Number_of_descriptors = Total_Data_Size / 1
If(Total_Data_Size / 128 == 0) Total_Number_of_Blocks++
If(Total_Data_Size / 1 == 0) Total_Number_of_descriptors++
Data Size transferred per descriptor = 1

Source address of each descriptor data = (1*Descriptor Number)
          + (128*Block Number) + Base address of source data

Destination address of each descriptor data = (1*Descriptor Number)
          + (128*Block Number) + Base address of Destination data

Data Size for last descriptor of last block =
          Total_Data_Size % Total_Number_of_descriptors
```

The above algorithm divides the total data size by 128 to calculate the number of blocks because the DMA controller of the FPGA possesses only 128 descriptors. The total data size is divided by 1 to calculate the number of descriptors because the Data size transferred per descriptor is 1 MB. In figure 3.2, 258 MB of data needs to be transferred by DMA write. 258 MB of data is split into 3 blocks. The first 2 blocks transfer maximum capacity of 128 MB using 128 descriptors each. The third block transmits 2 MB using only 2 descriptors.

In this mechanism of data transfer, buffers of 1 MB are used to transfer data. This basically involves two steps:

1. Transfer of data from user space to kernel Space
2. Transfer of data from kernel Space to FPGA memory

The maximum efficiency from the driver point of view can be derived if the above two operations are done in parallel. Transfer of data from User Space to Kernel space takes

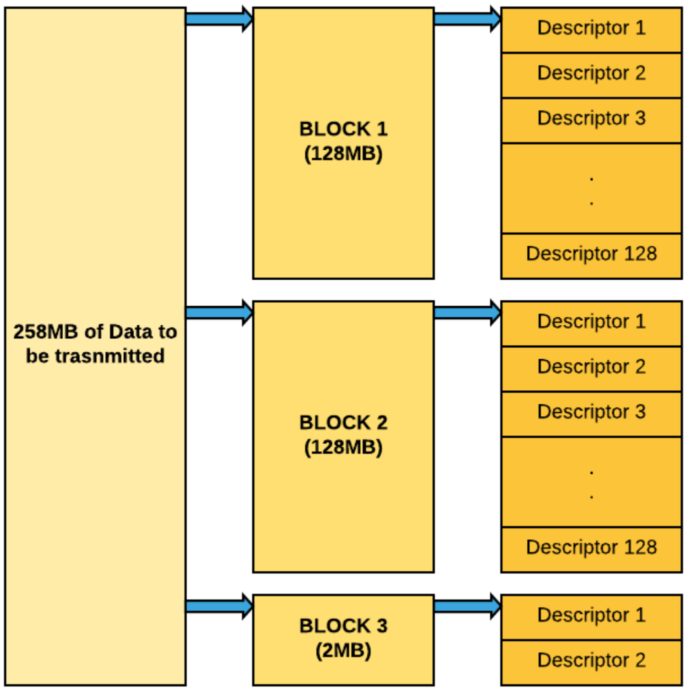

Figure 3.2: Splitting Data Chunks for DMA Transfer

around 11 percent of the total time to transfer data from the user space all the way to the FPGA memory [WsCTS03]. This 11 percent time is the maximum that the driver could save if an efficient parallel algorithm is implemented.

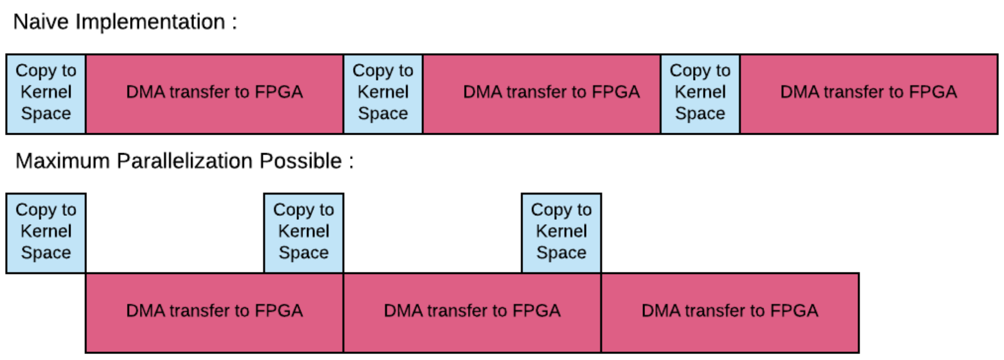

Figure 3.3: Maximum Parallelization Possible for DMA transfer

3.2.2 DMA Controller

The DMA controller present in the FPGA controls the read and write operations of the FPGA. The particular FPGA supports 128 descriptors each for read and write operations. The read and write are from the point of view of the FPGA. They are controlled by the read data mover and the write data mover. The read data mover transfers data from the PCIe address space in the root complex memory to the FPGA memory. The write data mover transfers data from the FPGA memory to the PCIe address space in the root complex memory. Initially, the driver should program the 128 descriptors and its contents in the PCI address space. The descriptors basically consist of information regarding the length, source and the destination address for each data transfer. the DMA controller works similar to Intel FPGAs as mentioned in [Ua17a]. In addition to the descriptors, the PCIe address space in the root complex memory should also have 128 status registers each of 1 DWORD. In the particular implementation, each DWORD is 4 Bytes. The status table consisting of 128 DWORDS. On completion of the task by each DMA descriptor, the status register corresponding to the descriptor byte is updated. The DMA controller has the capability to update the corresponding status register by a single 'Done' bit after completion of each and every enabled descriptor. This option can be enabled by setting the configurations in the DMA controller table. Polling can be done by the root host PC to check if the descriptor task is completed. In addition, the DMA controller sends an MSI interrupt after completion of all the enabled descriptors. Receiving MSI interrupts rather than polling will not improve the latency by a considerable amount but in turn, saves clock cycles of the host PC as it can avoid polling to check if the status register has been updated. The descriptor table is located in the PCIe address space in the root complex memory after the 128 status registers. The 128 status registers and the descriptors should be in a 32-byte boundary in the root complex memory [Ua17a].

The descriptors are numbered from 0 to 127. Therefore Descriptor would have ID 0, the second descriptor would have ID 1 and so on until the 128th Descriptor has ID 127. The RD_DMA_LAST_PTR is a location 0x010 in the descriptor table. A read from this register specifies the last ID of the descriptor that was read. To trigger more reads one should write the ID of the descriptors to be read. The user could trigger all the descriptors by writing the last ID of the descriptor to be enabled. To enable all the 128 Descriptors, the ID number 127 should be written into RD_DMA_LAST_PTR. For example, if the read ID at RD_DMA_LAST_PTR is 71 and the user needs to enable 3 more descriptors, the user should fill in the details of the Descriptors of ID 72 73 and 74. Finally, the user should write 74 to the location RD_DMA_LAST_PTR. The same is applicable for DMA write. The user should similarly use the RD_DMA_LAST_PTR register at location 0x110. It is important to note that the descriptors could have out of order completion i.e. Descriptor ID 72 could complete before Descriptor ID 73. The MSI interrupt is shot the moment ID 73 is completed. Therefore the MSI interrupts cannot be completely depended on for deciding whether all the descriptors have completed their tasks. Many commercial systems are based on out-of-order completion so as to optimize the access to host memory channels. This is especially the case when the descriptors transfer data of different lengths. To overcome this issue, the implementation of the driver has taken the following steps:

- All the descriptors except the final descriptor transfers data of the same length.

Address Offset	**Register**	**Description**
0x0000	RC Read Status and Descriptor Base (Low)	Specifies lower 32 bits of the 32-byte boundary read status and descriptor table in the root complex memory
0x0004	RC Read Status and Descriptor Base (High)	Specifies upper 32 bits of the 32-byte boundary read status and descriptor table in the root complex memory
0x0008	EP Read Descriptor FIFO Base (Low)	Specifies lower 32 bits of the 32-byte boundary read status and descriptor table in the FPGA memory
0x000C	EP Read Descriptor FIFO Base (High)	Specifies upper 32 bits of the 32-byte boundary read status and descriptor table in the FPGA memory
0x0010	RD_DMA_LAST_PTR	The user should write the ID of the descriptors to be enabled. When read, specifies the last descriptor that was requested by the user.
0x014	RD_TABLE_SIZE	Specifies the total number of read descriptors. Set to the value = number of descriptors-1. The default and maximum value is 127.
0x018	RD_CONTROL	0 - Update only the last status register after all reads are completed.1 - Update every status register immediately after completion of corresponding descriptorâ€™s task.

Table 3.3: Read Descriptor Controller format [Ua17a]

- The final descriptor always transfers data of a length equal to or less than the previous descriptor.
- In case of polling, the register RD_CONTROL at 0x0018 is enabled. This makes sure that each status register corresponding to each descriptor is updated with Done bit the moment the descriptor completes its task. However, this can be useful only in cases where polling is done.

Address Offset	Register	Description
0x0100	RC Write Status and Descriptor Base (Low)	Specifies lower 32 bits of the 32 byte boundary write status and descriptor table in the root complex memory
0x0104	RC Write Status and Descriptor Base (High)	Specifies upper 32 bits of the 32 byte boundary write status and descriptor table in the root complex memory
0x0108	EP Write Descriptor FIFO Base (Low)	Specifies lower 32 bits of the 32 byte boundary write status and descriptor table in the fpga memory
0x010C	EP Write Descriptor FIFO Base (High)	Specifies upper 32 bits of the 32 byte boundary write status and descriptor table in the fpga memory
0x0110	WR_DMA_LAST_PTR	User should write the ID of the descriptors to be enabled. When read, specifies the last descriptor that was requested by the user.
0x0114	WR_TABLE_SIZE	Specifies the total number of write descriptors. Set to the value = number of descriptors-1. The default and maximum value is 127.
0x0118	WR_CONTROL	0 - Update only the last status register after all writes are completed.1 - Update every status register immediately after completion of corresponding descriptorâ€™s task.

Table 3.5: Write Descriptor Controller format [Ua17a]

Address Offset	Register Name	Description
0x00	RD_RC_LOW_SRC_ADDR	Lower 32 bit address of data to be read in root complex memory
0x04	RD_RC_HIGH_SRC_ADDR	Upper 32 bit address of data to be read in root complex memory
0x08	RD_CTLR_LOW_DEST_ADDR	Lower 32 bit address of destination address in FPGA memory
0x0C	RD_CTRL_HIGH_DEST_ADDR	Upper 32 bit address of destination address in FPGA memory
0x10	CONTROL	Specifies the following information:Descriptor ID. Size of Data to be transmitted
0x14	Reserved	N/A

Table 3.6: Read Descriptor format [Ua17a]

3.2.2.1 Read and write DMA descriptor format

The read and write descriptor tables are present separately at different offsets. For the particular implementation, they are present at offsets 0x00000004 and 0x00002000 respectively in the FPGA memory. The read and write table have 128 descriptors each respectively of 8 DWORDS each. The locations of the status registers and descriptors in the root complex memory and the FPGA memory should be programmed initially into the registers in the DMA descriptor controller tables. The descriptor table is to be present at offset 0x200 from the status table base address. The DMA controller automatically adds this offset into the base address of status register entered in the descriptor controller table to find the descriptor table. Each descriptor can transmit a maximum of 1MB data. Writing the descriptor ID into the location in the descriptor controller table triggers the DMA read or write operation. Therefore precaution should be taken to write into the register only after initializing the controller table, status table and the descriptor table. Programming the controller table during DMA transfers isn't allowed. In addition, programming the controller table until all the previously specified number of descriptors is not allowed because it follows a FIFO model.

Address Offset	Register Name	Description
0x00	WR_RC_LOW_SRC_ADDR	Lower 32 bit address of data to be transferred from FPGA memory
0x04	WR_RC_HIGH_SRC_ADDR	Upper 32 bit address of data to be transferred from FPGA memory
0x08	WR_CTLR_LOW_DEST_ADDR	Lower 32 bit address of destination address in root complex memory
0x0C	WR_CTRL_HIGH_DEST_ADDR	Upper 32 bit address of destination address in root complex memory
0x10	CONTROL	Specifies the following information:Descriptor IDSize of Data to be transmitted
0x14	Reserved	N/A

Table 3.7: Write Descriptor format [Ua17a]

3.2.3 Algorithms for efficient DMA transfer

Once this is done, based on the previously selected performance level, the appropriate transaction algorithm is employed. The three different developed options for DMA transfer are the following:

- *Polling*
- *MSI - Minimal Performance*
- *MSI - Maximal Performance*

3.2.3.1 Polling

In this algorithm for implementation, 4 buffers are used for transmitting data from user space to kernel space. The number of buffers can be varied before loading the driver. The algorithm is implemented such that the number of buffers can be changed from 2 to 4 before loading the driver. The user has the ability to choose the number of buffers based on the memory constraints of the system. The implementation uses polling to determine if the user DMA controller has finished the task i.e. if the descriptor has completed the transfer of data. The DMA controller updates the value of the status register corresponding to the descriptor number on completion of the task. The CPU needs to keep polling this register to determine if the transaction is completed. In case of N buffers, the CPU copies 1 MB of data to each of the N buffers in the kernel space from the block of data from the userspace. The moment it completes writing to the (N-3)rd buffer, the DMA transfer by the FPGA is triggered. On updating the Nth status register with the done bit, the copying from userspace starts again to the buffers. This increases the efficiency of the DMA read operation by 9 percentage compared to the Naive implementation. This is the default configuration chosen on driver startup. Figure and depicts the timeline diagrams for DMA read and DMA writes respectively.

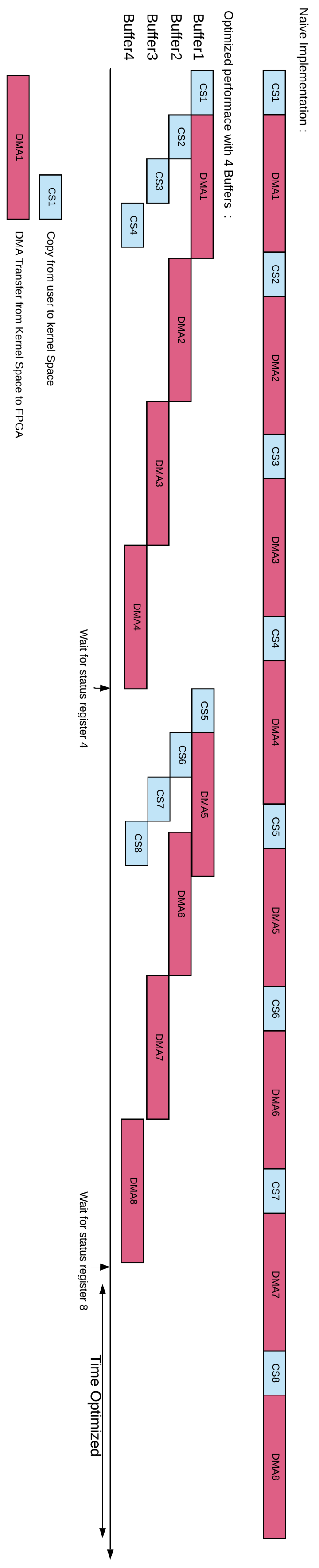
Naive Implementation :
CS1
DMA1
CS2
DMA2
CS3
DMA3
CS4
DMA4
CS5
DMA5
CS6
DMA6
CS7
DMA7
CS8
DMA8
Optimized performace with 4 Buffers :
Buffer1
Buffer2
Buffer3
Buffer4
Wait for status register 4
Wait for status register 8
Time Optimized
Copy from user to kernel Space
DMA Transfer from Kernel Space to FPGA

Figure 3.4: DMA transfer from user space to FPGA for *Polling*

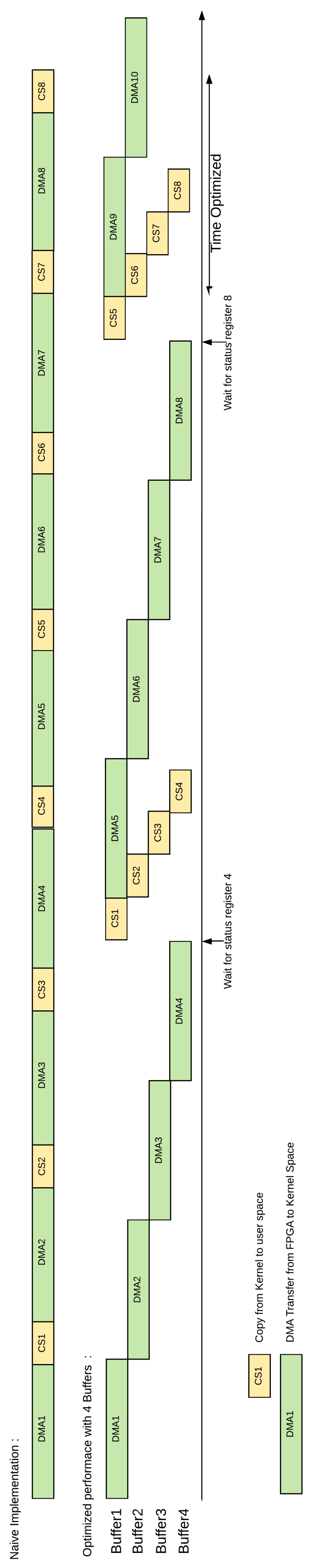

Figure 3.5: DMA transfer from FPGA to user space for *Polling*

3.2.3.2 MSI - Minimal Performance

The implementation varies from the *Polling* implementation in the following ways:

- Uses MSI interrupts rather than polling to determine if the DMA read/write task is completed.
- Uses 128 1MB buffers in the kernel space to perform the task

In this mechanism, the driver receives the offset, size and pointer to starting address in user space. The program splits the huge data chunk into blocks of 128 MB each. The 128 MB blocks are further divided into sub blocks of 1 MB each. The starting address of each of the blocks are programmed into the descriptors with appropriate lengths and offsets, Precautions should be taken such that the destination offsets are also programmed into the descriptors with appropriate destination offsets. The status registers should be initialized to 0. In case of DMA read, the data is copied from the user space to kernel space using the copy_from_user function. Once this is completed, the DMA transfer is enabled by triggering the RD_DMA_LAST_PTR in the descriptor controller table. The process goes to sleep after RD_DMA_LAST_PTR is triggered until it receives an MSI interrupt from the FPGA. Once the MSI interrupt is received, it performs the same operations with new block of data until the entire chunk is transferred. In case of DMA write, the chunk of data in the FPGA DDRAM memory is split into separate blocks. These blocks are subdivided into sub blocks of 1 MB each. The offset address of each of the sub block is again programmed into the descriptors. These are copied to the kernel memory by triggering the WR_DMA_LAST_PTR. Then the thread goes to sleep until it receives the MSI interrupt. The process is repeated until the entire chunk of data is transmitted. To parallelize the copy from user space to kernel space and transfer to FPGA, atleast 2 set of buffers is required. Using 2 sets of 64 buffers rather than a single set of 128 buffers increases the overhead as the descriptors and DMA controller needs to be reprogrammed after every 64MB transfer of data. In addition the thread has to be put to sleep and wait for MSI interrupts after every 64MB of data which is additional latency. The tradeoffs between efficiency gained to memory utilized should be considered. In applications where the memory utilization and CPU utilization are constrained, the particular implementation can be used. Figure 3.6 and 3.7 depicts the DMA read and write using MSI interrupts.

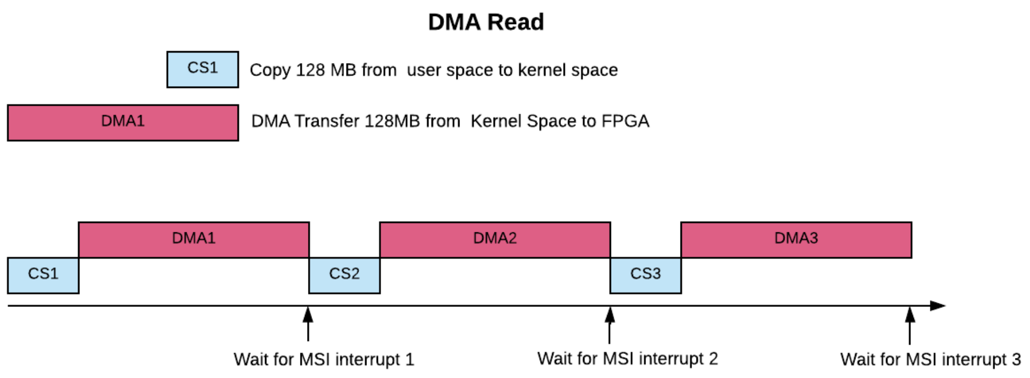

Figure 3.6: DMA transfer from user space to FPGA for *MSI - Minimal Performance*

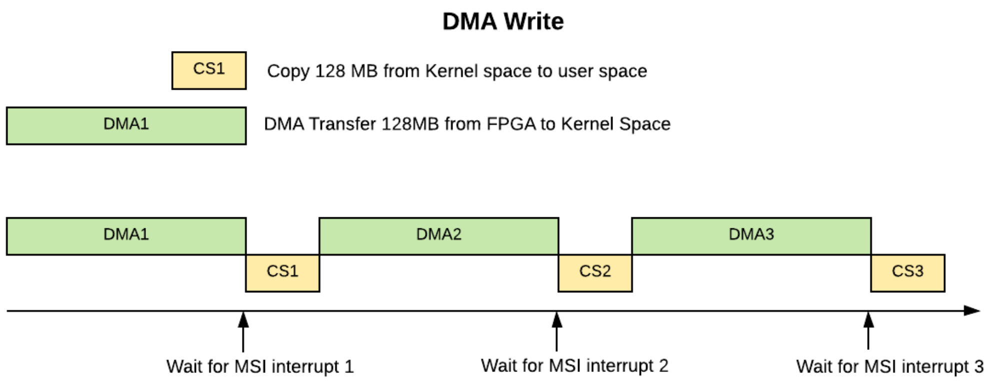

Figure 3.7: DMA transfer from FPGA to user space for *MSI - Minimal Performance*

3.2.3.3 MSI - Maximal Performance

The implementation uses 256 Buffers of 1MB each for DMA read and write. 256 buffers are used so that copy from user space to kernel space and transfer of data from kernel space to FPGA can be done in parallel. This saves the 10 percent time of total time taken to transfer data from user space to kernel space. It wouldn't be technically feasible to decrease the number of buffers, the FPGA by designed to transfer 128 MB with its provided 128 descriptors at one go. Therefore trying to decrease the buffer by any smaller amount and performing overlap wouldn't optimize much considering the overhead involved in parallelizing the code including waiting for MSI interrupts twice rather than once every 128 MB of data and reprogramming the descriptor controller and triggering the transfer. This is the reason the implementation is parallelized with exactly 2 sets of 128 buffers. In case of DMA read, the data is first split into smaller blocks of 128 MB. The block is copied to kernel space from user space. Then the Descriptors are programmed and finally the DMA controller is programmed and RD_DMA_LAST_PTR is triggered. The thread then moves on to process the next block of data and are then copied again from user space to kernel space. The FPGA parallely transmits the first block of data to the FPGA DDRAM. Once the copy of the 2nd block to kernel space is done, the thread goes to sleep until the FPGA completes its task and sends the MSI interrupt. On receiving the MSI interrupt, the RD_DMA_LAST_PTR is triggered again and the FPGA passes the 2nd block of data to the FPGA DDRAM. This process repeats again until the entire chunk of data from user space is transmitted to the FPGA DDRAM. Similarly for DMA write, the host PC, initially divides the data in the FPGA DDRAM into separate blocks. The FPGA DMA controller's RD_DMA_LAST_PTR is triggered so that the first block of data is transmitted. On completion the 2nd block of data is triggered. On completion of transfer of the first block of data to the kernel space, it sends an MSI interrupt to the host PC. On receiving the first MSI interrupt, the host PC transfers the 1st block of data to the user space from the kernel space. Likewise on receiving the 2nd MSI interrupt it copies the 2nd block of data from the user space to the kernel space. This process is repeated until the entire chunk of data is transmitted from the FPGA to the host PC's user space. Figure 3.8 and 3.9 depicts the DMA read and write operation for the implementation.

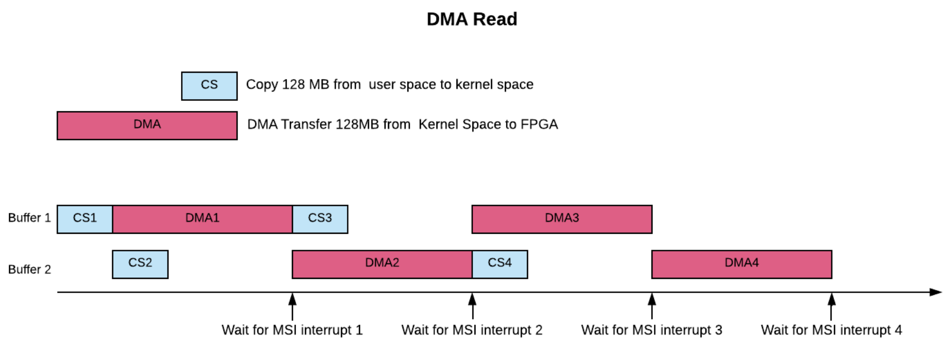

Figure 3.8: DMA transfer from user space to FPGA for *MSI - Maximal Performance*

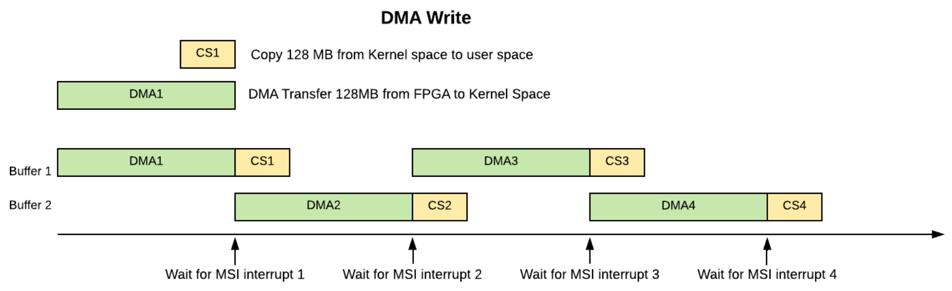

Figure 3.9: DMA transfer from FPGA to user space for *MSI - Maximal Performance*

3.3 MSI interrupts

A Message Signal Interrupt is a write operation from an external device to a predefined location in the CPU [RB05]. This write triggers an interrupt in the CPU. The MSI interrupt was initially first introduced in PCI 2.2. A new optimized implementation was also introduced from PCI 3.0 onwards called MSI-X. The external device can support both MSI and MSI-X. It is designed by specification and possible to use only either of the interrupts for any particular driver implementation. The device driver implementation uses only MSI interrupts and doesn't support MSI-X interrupts. The MSI interrupts are preferred over pin based interrupts because of the following reasons [Bot18]:

- Pin-based Interrupts are shared among several external devices. When an interrupt is received, that is, when a write is done to the special address, the kernel has to invoke the interrupt handler associated with each device and need to finally determine which device had sent the interrupt. This causes additional latency and undermines the idea of using interrupts to speed up the process.
- Functions in PCI devices support only a single pin-based interrupt [Bot18]. The function then has to analyze, query and determine what the interrupt was for. This increases the latency. However, the PCI devices can program the driver such that each MSI interrupt has a separate reason. Thus making the interrupt and subsequent decision making faster.
- Usually, the external device sends the data and then triggers the interrupt. But there is a huge possibility that the interrupt sent by the external device reaches the host PC before all the data reaches. This is a problem and the host PC needs to reconfirm the same. The PCI transaction rule states that the value of the special register should not be returned before all the data arrives in memory. In MSI interrupts, the interrupt generating write is a separate entity compared to the data write entity. Since the interrupt generating write sends a signal after the data write is complete, it is reaffirmed and need no further verification by the host PC.

From the Linux kernel point of view, the following flags should be set in the kernel config file [Bot18].

CONFIG_PCI_MSI:
This allows device drivers to enable MSI (Message Signaled Interrupts). MSI enable a device to generate an interrupt using an inbound memory write on its PCI bus instead of asserting a device IRQ pin.
X86_UP_APIC:
A local Advanced Programmable Interrupt Controller) is an integrated controller in the CPU. The local APIC supports CPU-generated self-interrupts.
CONFIG_IRQ_REMAP:
Supports Interrupt remapping for IO_APIC and MSI devices

The following function allocates the special address for the interrupt write in the host PC and enables the MSI interrupt [5018].

```
int pci_alloc_irq_vectors(struct pci_dev *dev, unsigned int min_vecs,
                          unsigned int max_vecs, unsigned int flags);
```

The device has the capability to request for a number of MSI interrupts. The maximum number allowed per device is 32 [Buc04]. It allocates MSI interrupts in the order of 2. Therefore 2, 4, 8, 16 or 32 interrupts can be requested by the device. However, the number of interrupts that are actually allocated to the device is decided by the kernel and is returned by the function. The function returns a negative value if no interrupts were allocated to the device. The driver should specify if it requires MSI or MSI-X capability as well during the function call.
The driver then should determine the IRQ number by using the following function [5018]:

```
int pci_irq_vector(struct pci_dev *dev, unsigned int nr);
```

When a PC receives the interrupt request (IRQ), it pauses its present running program and invokes an interrupt handler to run instead. An external device sends its interrupts through the IRQ line. The x86 system used in the implementation has a separate Advanced programmable Interrupt handler (APIC) that is integrated with the system. The APIC has 255 physical hardware IRQ lines each. Two devices are allowed to share the IRQ by using the IRQ shared flag. However, both the devices are not allowed to use them simultaneously. The external device derives all the information it requires about where and what signal to send as MSI interrupt through the MSI capability register. The register is configured by the kernel of the host PC during loading of the driver. The MSI capability register provides the following information to the device:

- Target address for interrupt write
- The number of MSI messages possible
- Value to be written into the address

3.3.1 MSI capability register

Every PCIe device has an MSI Capability register set within its configuration space. This is used for communication between the driver and the device to exchange the above-mentioned information.

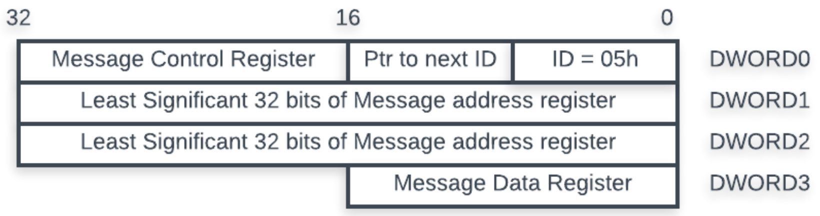
32
16
0
Message Control Register
Ptr to next ID
ID = 05h
Least Significant 32 bits of Message address register
Least Significant 32 bits of Message address register
Message Data Register
DWORD0
DWORD1
DWORD2
DWORD3

Figure 3.10: MSI capability register [RB05]

3.3.1.1 Capability ID

The capability ID for the MSI capability register is 05h and is hardwired and can only be read by the driver [RB05].

Finally, the API pci_enable_msi() is used to enable the MSI capability.

The values present in the current implementation is help. This shows that MSI interrupts were requested and help requests were allocated.

3.3.1.2 Message Address Register

The lower 2 bits of the address register is 0 by default [RB05]. The upper 32 bits of the address register will be set to 0 by the system software if the bit 7 of the control register is 0. If the 7th bit of the control register is set, it means that the device is capable of addressing 64-bit addresses in the host PC. The system software hence sets the upper 32 bits as well.

3.3.1.3 Message Data Register

The upper 16 bits of the device is always set to 0 [RB05]. The value in the Message Data Register is written to the host PC's memory whenever the device wants to send an interrupt. The lower 16 bits of the register varies depending on the number of MSI Interrupts allocated to the device. The lower 16 bits change such that it can convey the appropriate message to the host PC through its interrupt.

3.3.1.4 Message control register

Bit	Field Name	Description
7	64 bit address Capable	0 = Function can read only the lower 32 bits of the Message address register and is not capable of generating 64 bit addresses.1 = Function can read all the lower 64 bits of the Message address register and is capable of generating 64 bit addresses.
6:4	Multiple Message Enable	The system software writes the value. It reads the Multiple Message Capable register and checks if the requested number of messages can be allocated. Based on its decision it updates the register to indicate the number of messages are allocated.
3:1	Multiple Message Capable	The register value indicates the number of messages the device requests to be allocated. It should be a value of the power of 2 and maximum 32.
0	MSI Enable	0 = indicates MSI capability is disabled1 = indicates MSI capability is enabledReturns back to 0 when device is reset.
15:0	Reserved	Always 0

Table 3.8: MSI control register format [RB05]

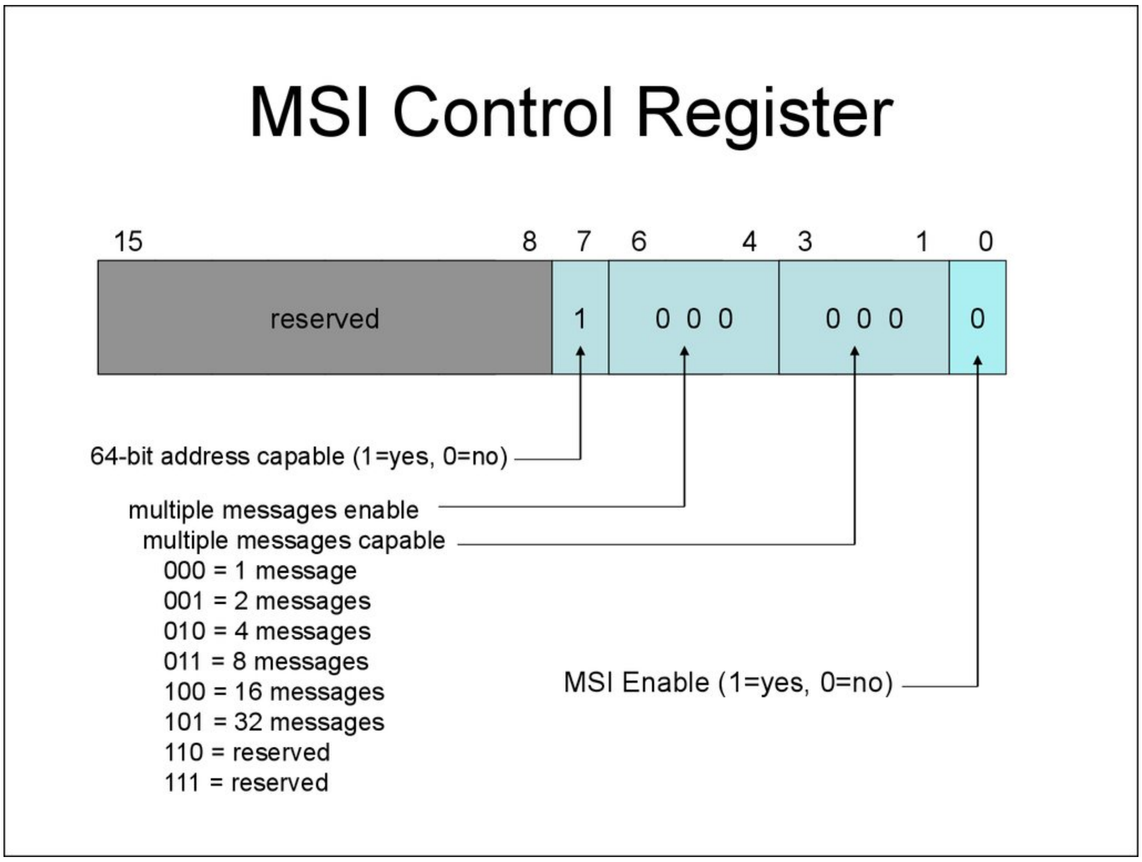

Figure 3.11: MSI control register [RB05]

3.3.2 Interrupt Handling

Programming the external device to send interrupts to the host PC isn't all that is sufficient. The host PC should be also be programmed. A software handler should be programmed in the host PC to perform operations on receiving an interrupt. The host PC should also be made aware of the external devices which would be sending interrupts. Else, the Linux kernel just ignores the interrupt. The Linux kernel maintains a registry of all the interrupt lines. It allocates an interrupt line to an external device on request and updates the registry. On removal of the device, it deallocates the IRQ line making it available for other devices. In some special cases, the IRQ line can be shared by multiple devices. An interrupt line can be reserved by the device driver on its initialization or only when it is required. Reserving an interrupt line during initialization is not advised if the number of IRQ lines are limited and the number of external devices is higher. But allocating and deallocating the IRQ line before and after every usage is too much overhead and decreases the performance of the system, Therefore, the driver is implemented such that the IRQ line is requested during initialization. Once the IRQ line is set, the PC should set up asynchronous notification. There are several steps to be taken to do that [JC05]:

1. The user needs to allow asynchronous notification from the input file. The process invokes the F_SETDOWN command using system call. This ensures that the kernel

notifies the process on receiving an interrupt. The kernel saves the process ID in filp->f_owner. Without this step, the kernel just ignores the interrupt.

2. The next step is to actually enable the asynchronous notification for the process by setting the FASYNC flag by using system calls.
3. The final step is to request delivery of SIGIO signal whenever it receives an interrupt. The interrupt data is sent to the process ID stored in the pointer filp->f_owner.

The 2 functions invoked are as follows [JC05]:

```
int fasync_helper(int fd, struct file *filp, int mode,
                                    struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

Fasync_helper is used to notify the kernel that the particular process should be interrupted by the kernel on receiving an interrupt signal by the particular external device. It adds the process into the list of processes to be notified. The kill_fasync signals the processes that have been set by the Fasync_helper function. Its arguments include the band which is POLL_IN for the particular implementation and the signal to send which is POLL_IN or POLL_OUT based on whether it is an interrupt received on DMA read or write.

3.4 PIO read/write

Programming Input Output (PIO) is a mechanism to control and configure the FPGA from the host PC. In PIO, the software written in the PC is responsible for transferring data to and from the FPGA. PIO data transfers can be done by Port mapping or Memory mapping. The implementation performs PIO using memory mapping. MMIO refers to mapping wherein the memory is allocated in the address space usually used for program and data. At the low level, it implements the data transfer by using instructions like LOAD, STORE, etc. The PIO read/writes are much slower than DMA read/writes.

The FPGA has 16 sockets each of which have a unique ID. These IDs are located at 0x8 and 0x10 offset from the starting address of each socket. The lower addresses are located at 0x8 and the upper addresses are located at 0x10. Each of the sockets also has a test register at 0x28 to which PIO read and writes can be done. Each of the sockets comprises of 8192 bytes. Therefore to access the UUID addresses the following formulas can be used:

```
Accelerator Slave UUID low addresses: N*2048 + 0x8
Accelerator Slave UUID high addresses: N*2048 + 0x10
Test Register = N*2048+0x28
where N=Accelerator Socket Number
```

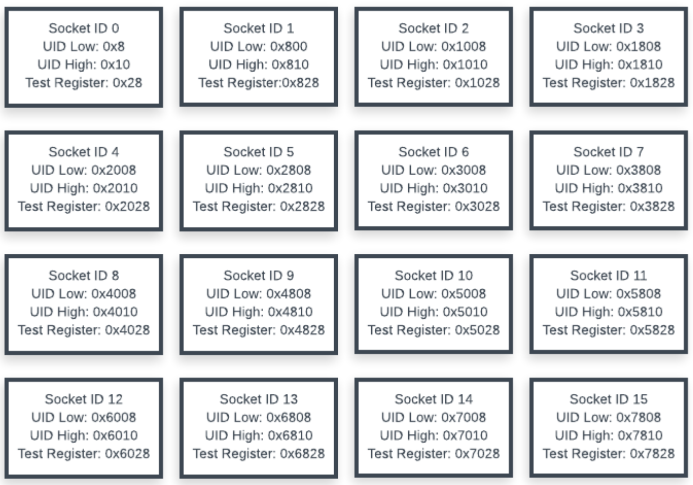

Figure 3.12: FPGA sockets for PIO read/write

3.5 Real-time Performance Altering

The driver has the ability to perform several levels of optimization based on the requirement. Many at times, the driver needs to work in environments where the memory is a major constraint for instance when the server has limited memory or has many applications working in parallel. In such cases taking a toll in the rate of data transfer is allowed. But in other cases where the memory is not a constraint like in data centers where high-performance hardware and huge memory RAMs are completely available for the sole purpose, performance should be maximal. Therefore the driver has three implementations for DMA transfer:

1. *Polling*
2. *MSI - Minimal Performance*
3. *MSI - Maximal Performance*

There are instances even in data centers where the maximum performance allowed per application varies greatly depending on the time of the day. For instance, during the daytime, there could be larger customer care calls and services being used. Therefore host services and internal services should give space to the customer care services for better output. During the night time, when the customer care services have decreased, the host and internal services can work using maximum capacity. Therefore real-time performance altering is an indispensable tool in today's industry where time, cost and performance are trivial. The driver works in *Polling* mode by default on being loaded. The user can use ioctl commands to change the performance level. In addition, the user can use the ioctl command to discover the level of operation currently in. Using grafana, the user has the ability to view the performance level remotely from any location which has access rights to the FPGA. The following scenarios were considered when building the design:

- The user may use multiple applications to connect to the driver at the same time.
- The user may connect to device files of multiple FPGAs at the same time i.e. the user may simultaneously access multiple FPGAs.

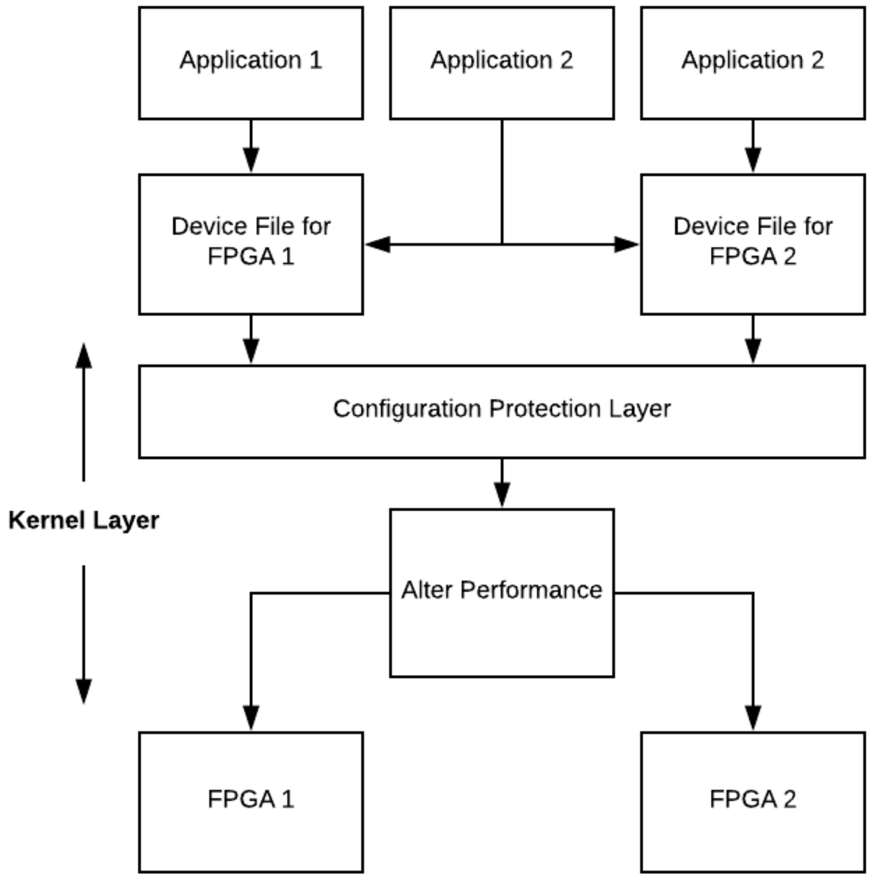

Figure 3.13: Work flow of management layer software

Therefore precautions should be taken such that no alteration is possible while a DMA read or write is being done by any of the applications or FPGAs. Care should also be taken such that simultaneous alterations of the performance level does not take place from multiple applications. These issues have been solved by introducing a new layer called the configuration protection layer in the kernel space. This layer makes sure that all the above conditions are fulfilled and changing configuration will not disrupt the driver or affect any other application. Once this is verified the application or user is given access to the change any configuration. A unique variable is introduced for universal synchronization. This variable "dmaspeed" can be accessed only with mutex locks. Conditions also have to be fulfilled by the driver and the application for the user to be allowed to make alterations. Once these alterations are made, the driver is updated and the mutex is unlocked.

3.6 Management Software for parallel programming

The user APIs initially developed work serially. Once the user issues a read or write, it has to wait until the task is completed. The user can operate a second FPGA from a completely different application, but not from the same application. This, however, turns out to be inefficient in cases where the user needs to access many FPGAs from the same application at the same time. Secondly, according to traditional approaches, the user application needs to reach all the way to kernel level to figure out that the FPGA is already being used. The APIs would be more efficient if the user application can determine the state of the FPGA from the user space itself. Thirdly, the traditional approach uses serial programming. Once the user issues a DMA read or write, the process has to wait until the task is complete to perform other tasks. All the above drawbacks of the traditional driver show the need for a new novel mechanism. This implementation deploys an additional layer at the userspace level to eliminate all the above pitfalls of the orthodox drivers.

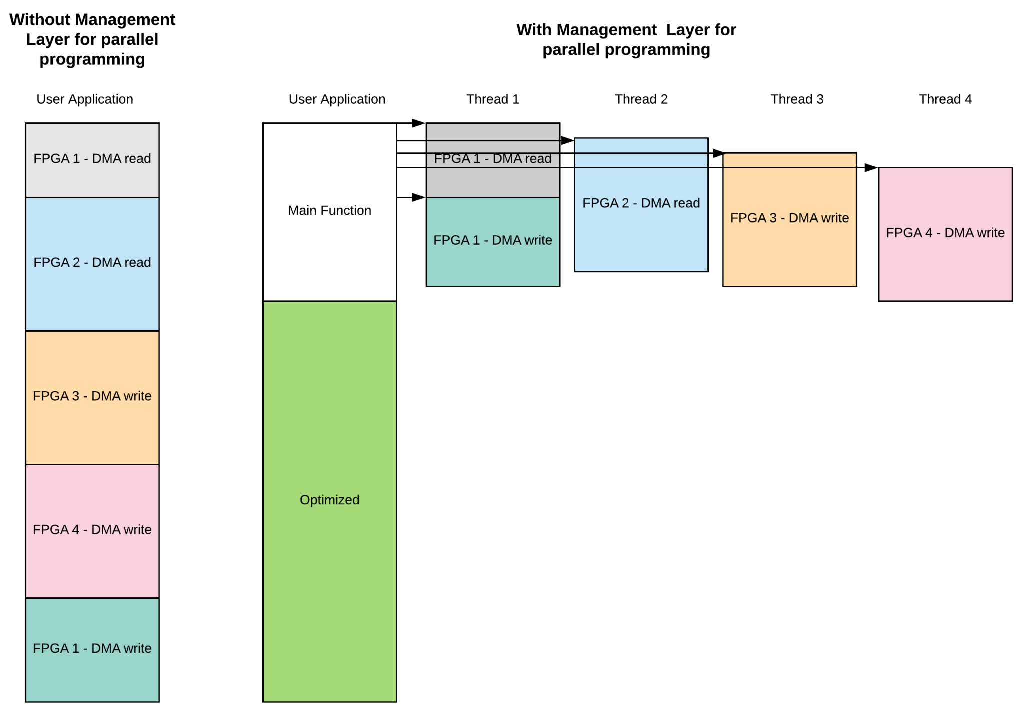

Figure 3.14: Optimization Achieved through management layer for parallel programming

The top management layer works in such a way that when a DMA read or write is issued, the parent kernel which is linked to the main function initializes a new thread which foresees the subsequent checks and issues the DMA read or write. The new thread performs the following operation:

1. It verifies if the FPGA is already in use or is executing some other task issued by the same application.
2. If the FPGA is not already being used by the current application, it creates a lock so that the current application cannot issue another task to the FPGA.
3. The thread then issues the command and passes on the information to the device file and then to the kernel driver level.
4. The driver checks if the FPGA is being used by any other application.
5. The driver verifies that the driver configuration file is stable i.e. all the configuration changes have been implemented and executed.
6. DMA read or write is executed as per command.
7. The management layer waits for the execution to be completed
8. Once the DMA transfer is completed, it removes the inuse flag for the FPGA so that any other application can use the FPGA.
9. The thread changes its ’in use flag’ at the user level so that the parent application can use the FPGA.
10. It removes the mutex lock and self-destructs the thread on closing the device file.

The parent thread can simultaneously work on other tasks or other FPGAs while the above actions are executed. It can decide if the transfer is complete by checking the inuse flag of the FPGA. Pthreads are used for the parallel implementation. In the traditional approach when the device file is opened to access the driver, the user is returned the file number which is used for all subsequent actions and references. However in this case when the device file is opened successfully a pointer to a structure is returned. The structure is as follows:

```
struct  fpgacontrol {
  ssize_t file, len;
  pthread_mutex_t lock;
  u_int32_t inuseflag, testoption;
  pthread_t userthread;
  u_int64_t offset;
  char  *buf; u_int64_t off;
};
```

This structure contains all the information that would be needed by the user application. The variable "file" contains the file number. It contains the threads which are already initialized so that the user does not need to initialize and destroy them for every DMA read or write. The variable inuseflag can be polled to check if the FPGA is currently being used by the FPGA. ’inuseflag’ can also be polled to check the completion of a task by the

FPGA. Each of the device file opened for each FPGA in each application would have a unique 'fpgacontrol' structure linked to it.

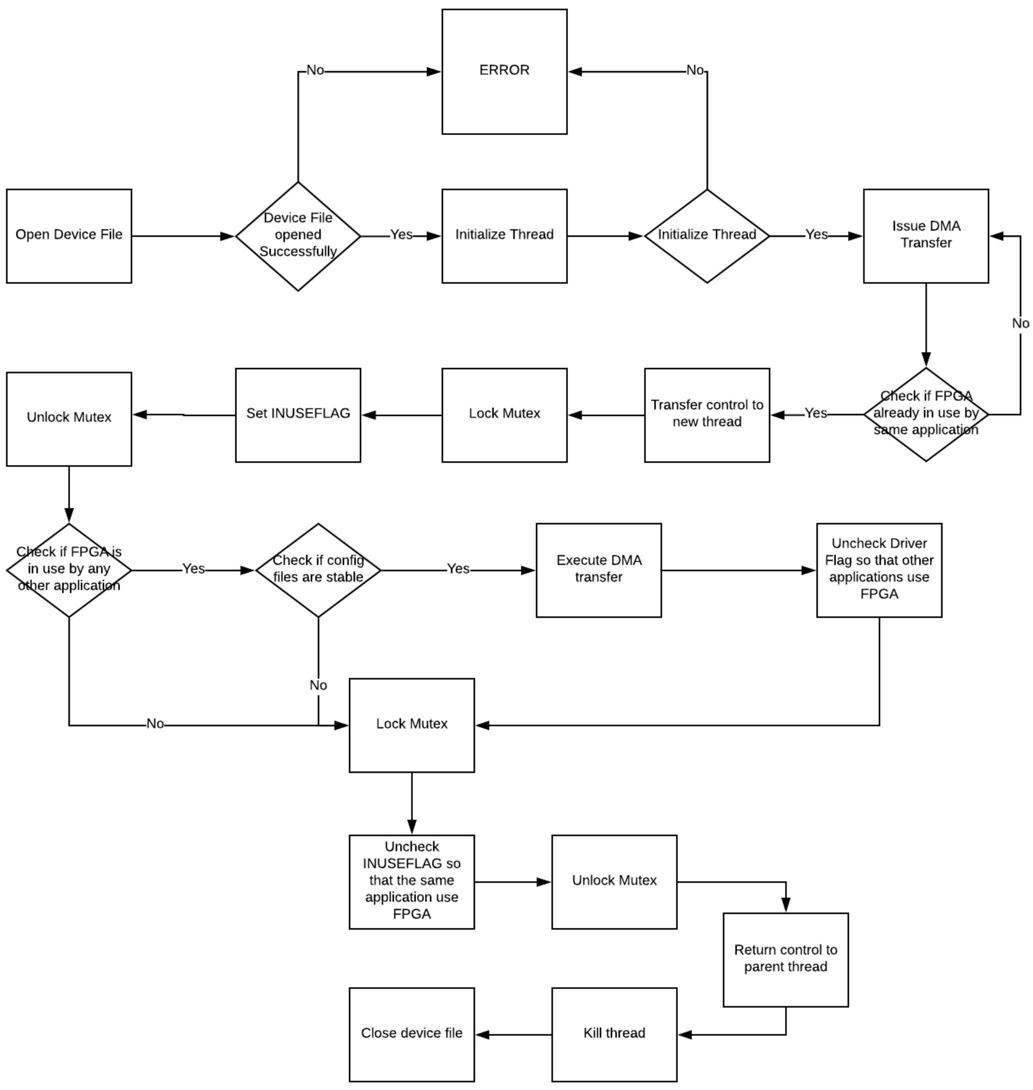
ERROR
No
No
Open Device File
Device File opened Successfully
Yes
Initialize Thread
Initialize Thread
Yes
Issue DMA Transfer
No
Check if FPGA already in use by same application
Yes
Transfer control to new thread
Lock Mutex
Set INUSEFLAG
Unlock Mutex
Check if FPGA is in use by any other application
Yes
Check if config files are stable
Yes
Execute DMA transfer
Uncheck Driver Flag so that other applications use FPGA
No
No
Lock Mutex
Uncheck INUSEFLAG so that the same application use FPGA
Unlock Mutex
Return control to parent thread
Kill thread
Close device file

Figure 3.15: Work flow for Management layer

3.7 Acquisition, storage and display of FPGA Parameters

Huge Data centers have thousands of high-performance devices that work round the clock to fulfill the services. With more power comes more responsibility. Efficient management and control of these devices is indispensable for the smooth functioning of the data centers. This needs appropriate tools to procure, store and view the data. The implementation uses a combination of telegraf, influxdb and Grafana to fulfill the above mentioned needs of the FPGA in real-time from any location. It presents a user-friendly GUI interface for easy manipulation and control of the data.

3.7.1 Telegraf

Telegraf is InfluxData's open source server for collecting data of the connected devices and server performance [2218]. This is the first piece of the collection, reporting, storage and analyzing of data. It comprises of various plugins to collect data about the CPU, pull metrics from the system, third-party APIs or read metrics from external devices. It has the capability to send data to different services or datastores including InfluxDb, Datadog, Kafka, Graphite, Labrata and many others. Telgraf is entirely written in GO and compiles into a single binary. It is useful as it has no external dependencies and therefore can be used out of the box. The implementation creates a container which can be easily installed on any server and used on the go. It has minimal footprint and new input and output plugins can be easily added by the user dynamically. It has a configuration file in which the user needs to specify the desired sources of data and the destination database to which the data needs to be sent. On installation, a default configuration file is deployed in /etc/telegraf . The file has to be configured based on the user's requirement. For the implementation, two plugins have been used. The output data formats of telegraf basically consist of four parts:

1. Measurement name
2. Tags
3. Fields
4. Timestamp

These four parts are exactly similar to InfluxDB Line Protocol. The protocol is a text-based format that lays the ground rules for writing into the InfluxDB [2118]. Each line elucidates a single point. Multiple lines can be written at once. But they need to be separated by a newline character. The line protocol format consists of three parts:

```
[key][field][timestamp]
```

Each of the parts is separated by spaces. Timestamp need to not be entered by the user. In case of no timestamp, telegrapf automatically includes the timestamp with nanosecond precision. Key is the name of the measurement. The user can add tags to the key to enable search query and segregation. Fields are metrics or measurements associated with the key. Each point should have atleast one measurement. Each of the measurements should be separated by commas. Field values can be of any of the four types. The user doesn't have to specify the type initially. On entering the first sample, telegraf automatically identifies the field and sets the data type. All subsequent fields should have the same datatype. Data types can be any of the following:

1. **Integers**: numerical values that are followed by an 'i' (example: 24i, 273i)
2. **Floats**: Floats are numerical values that are not followed by a trailing 'i'. Any numerical value that does not contain an 'i' are treated as floats. (example: 75, 32.1, 1)
3. **Boolean**: Indicate True or False. Can have any of the following formats: t, T, True, TRUE, FALSE, False, f, F.
4. **Strings**: All field values surrounded by double quotes.

The timestamp as mentioned is automatically added in nanoseconds precision. However, it can work in precisions ranging from hours, minutes, seconds, milliseconds or microseconds. Tags are optional. Tags are faster on querying compared to keys and therefore it is advantageous to use them. Tags are indexed while fields aren't.
Using the above line protocol the value entered by the telgraf into the InfluxDB is as follows:

Example 1:

```
esgfpga,FPGA_NUMBER=fpga1 temp=76,fpga_performance_level=0,
fpga_read_throughput=0.000000,fpga_write_throughput=0.000000
```

Example 2:

```
esgfpga,FPGA_NUMBER=fpga2 temp=76,fpga_performance_level=0,
fpga_read_throughput=0.000000,fpga_write_throughput=0.000000
```

Esgfpga is the name of the database to which the data is sent. FPGA_NUMBER is used to distinguish between the different FPGAs enumerated by the driver. fpga_performance_level, temp, fpga_read_throughput, fpga_write_throughput are the different metrics that are passed on. Fpga_performance_level conveys the performance level that the FPGA is currently working on. Temp metric displays the temperature of the FPGA. fpga_read_throughput and fpga_write_throughput convey the rate at which the DMA completed its last DMA read and write respectively.

Telegraf, in addition, provides an option to use an aggregator. The aggregator is rightfully placed in between the input and output sections of telegraf. Originally data passes only through the processor. The processor process the measurements that it derives from the input. It filters, transforms and decorates the data and immediately emit the data results. Aggregators process the data and provide a more refined result. It can be used for computing parameter mean, standard deviation, total, maximum or minimum over a period of time that the user presets. This preset time gap is called the period. Many at times, the user just needs to know this period data and not every metric as such. So if the user decides to view/store only the aggregated values and not every metric, the configuration can be set as such. However, the historical data of raw metrics is not supported.

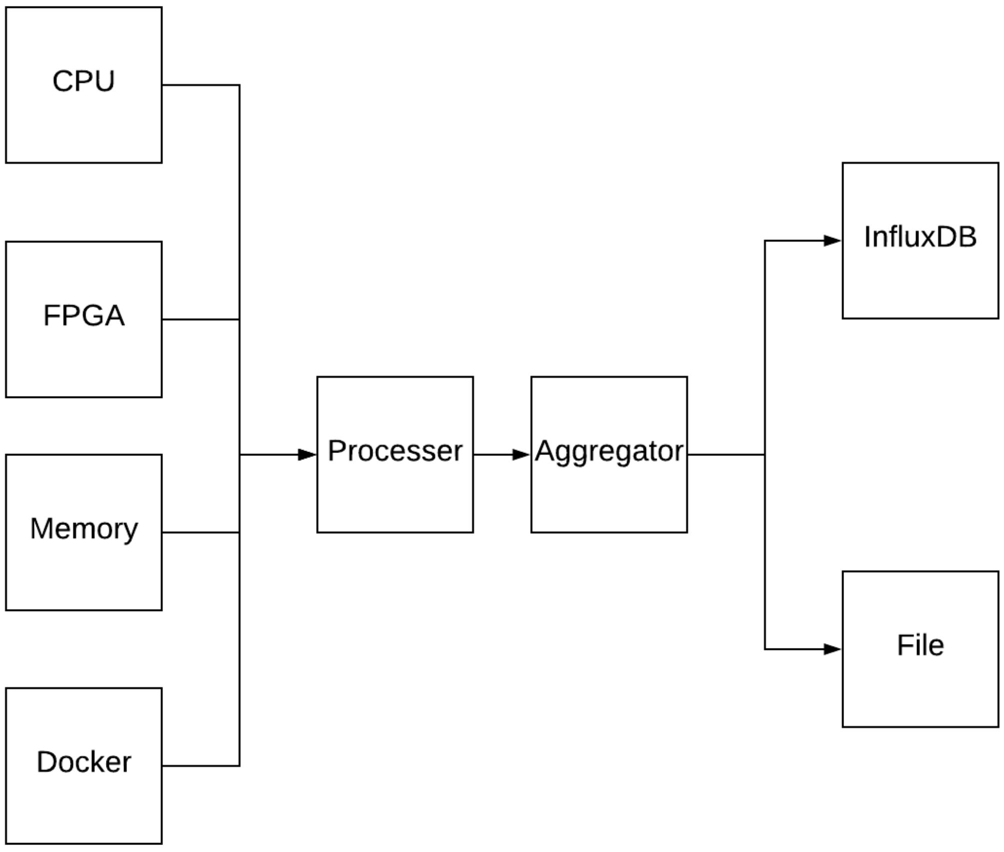

Figure 3.16: Work flow of the complete experiment.

The telegraf uses the exec plugin to read data from the FPGAs. The exec plugin executes a file every preset period of time. This file is designed such that it executes a user application. The particular user application to is programmed to determine the number of devices that have been successfully enumerated by the device driver. Then, it reads the FPGA measurements of each of the FPGAs that is on the loop. It obtains the ID of the FPGA, performance level, temperature, DMA read and write rates. Finally, the obtained data is converted into the InfluxDB line protocol format and written into separate data files. These files serve as the input material for the telegraf to process. The Data is overwritten into the separate data files of each FPGA after a period of 5 seconds so as to not overload the memory of the server.

3.7.2 InfluxDB

InfluxDB is a database that is optimized for using time series data [2118]. This includes sources like financial transactions, distributed sensors, device metrics, etc. Databases that have a lot of time-related data have more 'creating and reading of data' rather than 'update and destroy of data' behaviors. InfluxDB is targeting this niche market of 'creating and reading of data' and is particularly optimized only for the purpose. InfluxDB is useful for performing real-time analysis on a large amount of data much more quickly compared to SQL which is more broadly used. SQL too can operate on time series data, but research has shown that InfluxDB is quicker and easier for the purpose. An advantage of the InfluxDB is the developer need not specify the schema up front. It can be changed or updated in real time. For instance, as of now, the FPGAs has the ability to display temperature and performance measurements. But in future, when the FPGA is capable of updating other parameters like voltage, power consumption, etc, the schema can be updated dynamically. In addition, InfluxDB has the following advantages:

- Influx Data is out of the box solution. Therefore the time of development is lower.
- Easy to use tools to find value in the data by identifying patterns and predict the future.
- Millions of writes per second and clustering of data to avoid failure of any sort.

InfluxQL is an SQL like language for interacting and updating the InfluxDB. InfluxDB is similar to SQL as follows:

- Replication policies and continuous queries are identical to that of SQL Database.
- An influxDB measurement table called 'foodships' is similar to an SQL table
- InfluxDB tags are exactly the same as indexed columns in SQL Database
- InfluxDB fields are similar to unindexed columns in SQL Database.
- Points in InfluxDB is similar to rows in SQL.

Therefore a user with experience in SQL can easily grasp the methodologies of the InfluxDB with minimal effort. InfluxDB, hence, provides the advantage of an enhanced time series database with little effort from the developer.

3.7.3 Grafana

Grafana is an online analytics platform tool that can be used to query, visualize, alert and understand the time series data metrics irrespective of where it is stored [2018]. It allows the developer to create and share dashboards with informative and user-friendly GUIs to enable efficient data understanding. Grafana supports different storage backends. Graphite, InfluxDB, Prometheus, Elasticsearch, Cloud and graphite are officially supported. Each Data source has a separate Query Editor which supports all its specific features. Grafana has the ability to combine metrics from different sources to a single dashboard. It can also deploy its metrics to different organizations with each organization having its own limited access to the metrics. Each organization can have multiple users. A user can belong to different organizations as well. Each user can have different levels of privilege within the same organization. Panel is the basic block of any dashboard. Each panel has its own query editor so that the user can extract the right amount of data and build the perfect visualization to suit the needs. It consists of basically four types of panels: Graph, Singlestat, Dashlist and text. Graph allows the user to view the metrics based on a timeline. Graphs are used in the implementation to display FPGA DMA read and write throughput and temperature of the FPGA. Singlestats are used when the user needs to extract the data and show a single status. Singlestats is used to display the performance level of the FPGAs.

3.7.4 Overall structure

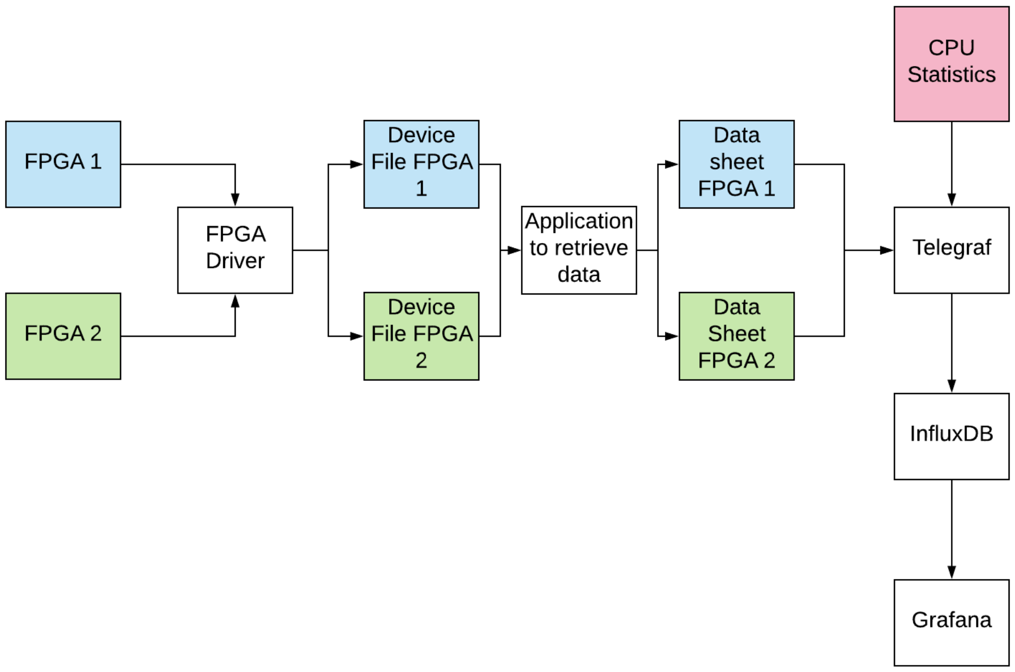

Figure 3.17: Design flow of user application's access to FPGA

The following flowchart shows the overall structure. The user application specifically designed to obtain data regarding the FPGAs retrieve the data by accessing the device file of the FPGAs. It writes the data into data sheets specifically for the FPGAs. Telegraf takes in this information from the data sheets and processes it to the right format. It also retrieves information regarding the server performance statistics and passes it on to the remote database. The Database InfluxDB stores the data in the specified format. The user form any part of the world can access this information stored in the database using its online analytic platform.

4 Final Design and Workflow

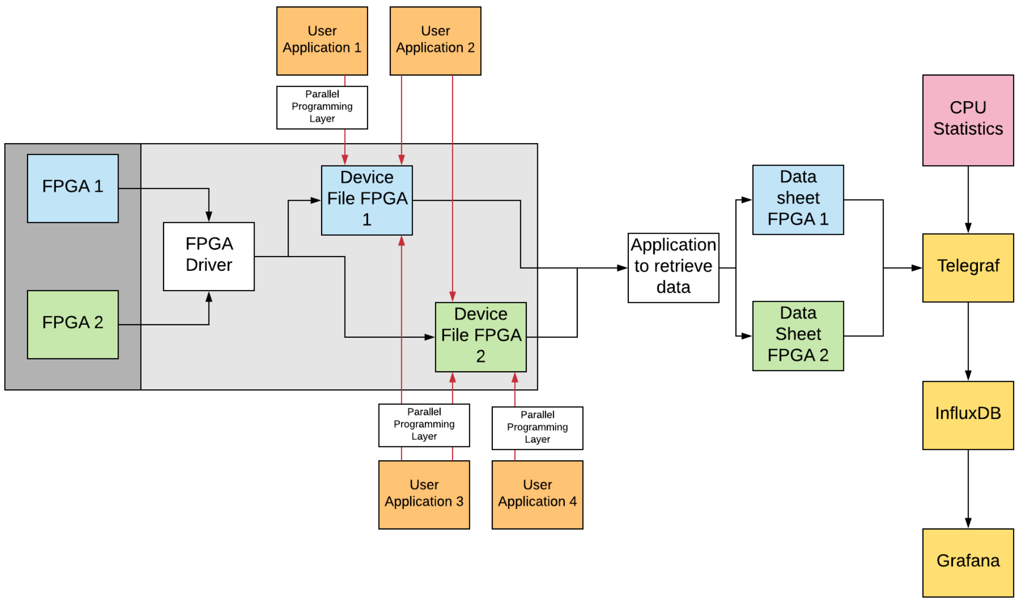

Figure 4.1: Work flow of the complete experiment.

The section portrays the final blend of all the above features put together. On loading the driver, it detects and enumerates the FPGAs connected to the system through PCIe. These devices are then labeled and device files are created. The user can access these FPGAs from the user space through the device files by using the opendriver functionality. Once opened, it can connect, control and transfer data to the FPGA from the user space or vice versa. It can also control the performance level of the driver depending on the requirements. As the figure shows, the user can choose to use serial or parallel programming APIs which are provided in the package. Based on the chosen API, the user can control multiple devices in parallel or cut out the overhead and use serial programming. Many applications can access the FPGA simultaneously at the same time. Applications can run with or without the top management layer functionality which enables parallel programming.

The driver package comes with an additional application which detects all the FPGAs that are available to the system. This application stores values like the temperature, performance levels, read and write throughput to separate datasheets. these datasheets are refreshed periodically so to avoid occupation of too much space in the system. The telegrapf application consumes the data present in the data sheets and processes them into the right format and passes them on to a remote server. This is done every few seconds. This period can be predefined by the user. In addition, the telegraf also sends CPU usage statistics to the remote database. The remote database InfluxDB stores and saves the data. Grafana uses the data from the InfluxDB to create a user-friendly GUI for the user to track the performance of all the FPGAs and the server remotely. Alarms are also set to warn the user in case of disruptions or unusually low performance or high temperature.

5 Test Evaluation and Results

5.1 Test environment

The FPGA driver needs to be tested to make sure that it works sustainably in all kinds of situation. Tests are done to make sure everything from the FPGA, PCIe, host PC and application are able to synchronously work together in harmony. The implementation comes with a large number of test APIs out of the box so that the user can use them based on the requirements. The test APIs are designed to be robust and modular. Separate test APIs have been designed for serial programming and for parallel programming. Both the set of APIs employ the same testing methods but have different interfaces to the user application. The following APIs are included in the test package:

- **PIO UUID Address Test**
  The test reads the UUID low and high addresses present at the offset locations 0x08 and 0x10 respectively of all the 16 sockets and compared them with the preset values. These preset values are predefined by the user. It alerts the user in case any of socket's UUIDs don't match.
- **PIO read-write test**
  Each of the 16 sockets has a test register at offset address 0x28. The test writes a random value and reads back the value using PIO write and read. It compares if the values match and raises an alarm otherwise.
- **Basic DMA test**
  The socket creates a string of random lengths ranging from 64 bytes to 2GB. It writes the value to a location in the first DDRAM. Once the DMA write is completed, it performs a DMA read of the exact same length from the exact location. Finally, it compares if the read and written values are identical.
- **Multiple DDRAM DMA test**
  The FPGA consist of 4 DDRAMs. Reading and writing to each of the DDRAM one after another involves a more complicated hardware level design. This sustainability is tested by reading and writing into different DDRAMs one after another randomly with data of different lengths. It then compares if the values written into and read from each of the DDRAM randomly are identical.
- **Marathon DMA test**
  This tests the sustainability of the algorithm involved in splitting the data chunks

into different blocks, sub-blocks and then finally into 1MB data. This algorithm of splitting needs to be tested. The best case would be to test the DMA read and write for all sizes of data. The minimum resolution of the DDRAM memory access is 64 bytes. Therefore this test writes all multiples of 64 bytes into DDRAM and reads the same amount of data back into the userspace. The user needs to define to what value the tests should be done. It then performs basic DMA tests with 64 bytes, 128 bytes, 256 bytes and so on until the predefined value.

- **Complete FPGA test**
  This test includes PIO UUID address test, PIO read-write test and multiple DDRAM DMA tests for each performance level. It is recommended that the user performs just this one test which includes the entire package before actually running the device.

5.2 Test Results

5.2.1 DMA throughput

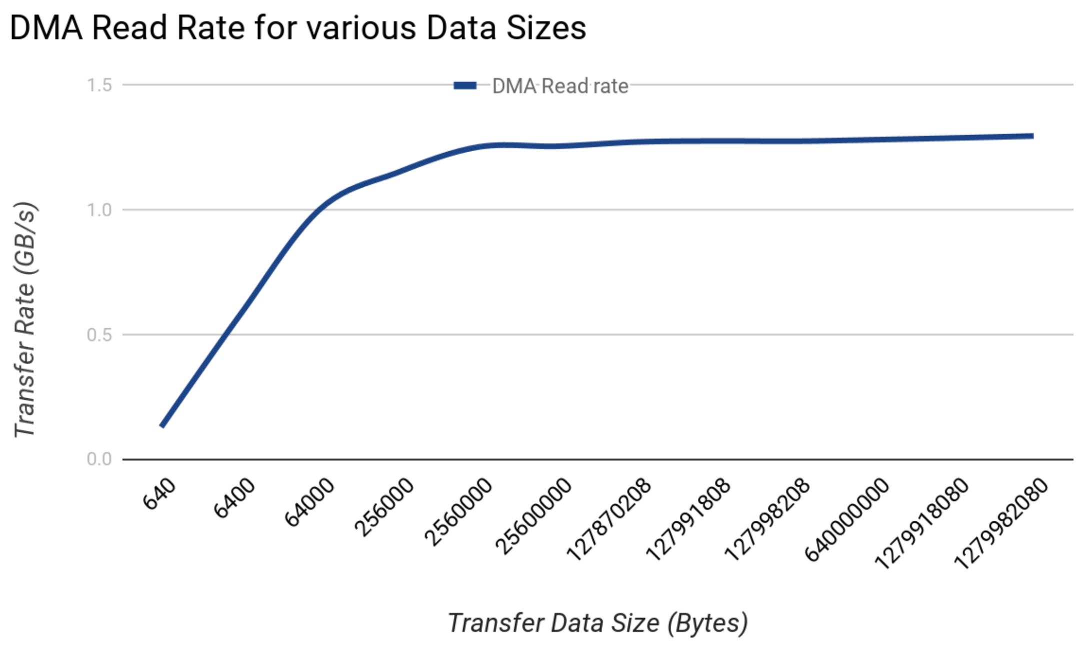

Figure 5.1: DMA read throughput for different Data Sizes

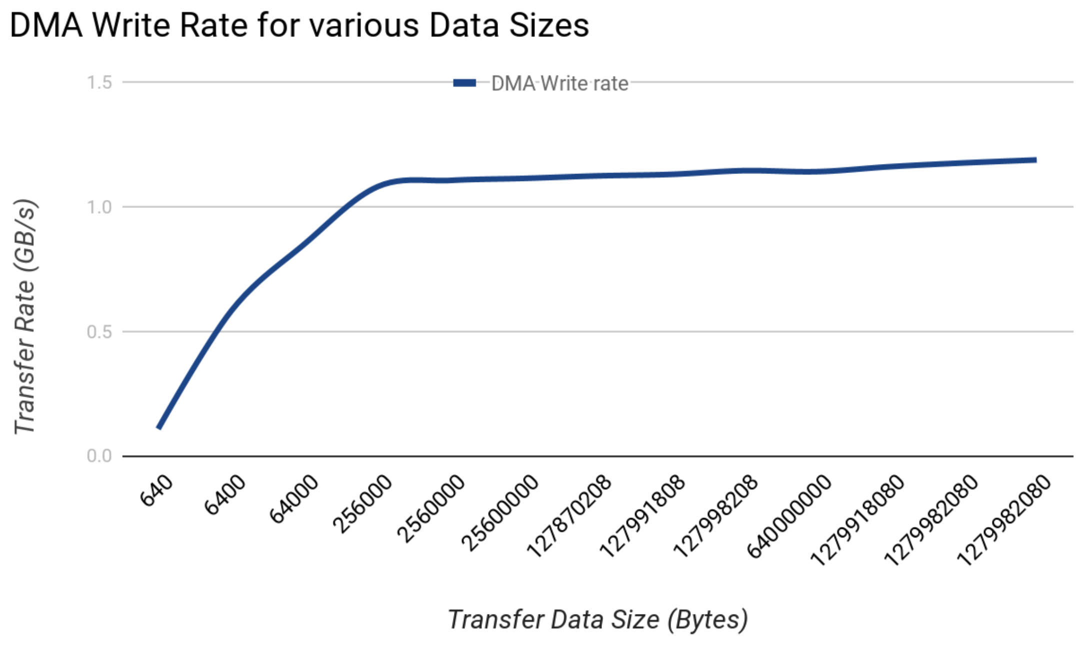

Figure 5.2: DMA Write throughput for different Data Sizes

Robust testing methods were used to enhance and improve the driver. Many levels of testing during the initial phase let the developer identify issues with the hardware level and software level codes. These bugs were rectified and tested over and over until sustainability was proven. The previously mentioned test APIs were used in different situations to track and analyse the driver systems.

The area of usage and time latency are the two main factors used for testing the performance of DMA transfers as shown in [SS17]. It was noted that the read and write throughput rate varied depending on the number of applications running on the system and other background activities. Therefore the results shown may vary slightly. The throughput rate for DMA read and write was highly dependent on the data chunk size that was transmitted per command. Figure 5.1 and 5.2 depicts the same. The major amount of time consumption took place for setting up the descriptor and the descriptor controller values. Therefore this is a clear proof that increasing the number of descriptors could significantly improve the performance of the driver. The codes are written such the number of descriptors can be varied based on requirement. Thus the codes are reusable. It can be seen that at 128 MB transfer size, it hits the maximum throughput and remains constant henceforth. This is because the FPGA's descriptor controller can use only a maximum of 128 descriptors at a time. Therefore after every 128 MB data transfer, the driver needs to reprogram the status register, DMA controller and the descriptors. This pays a major toll on the throughput.

5.2.2 IOCTL vs Device File Functions

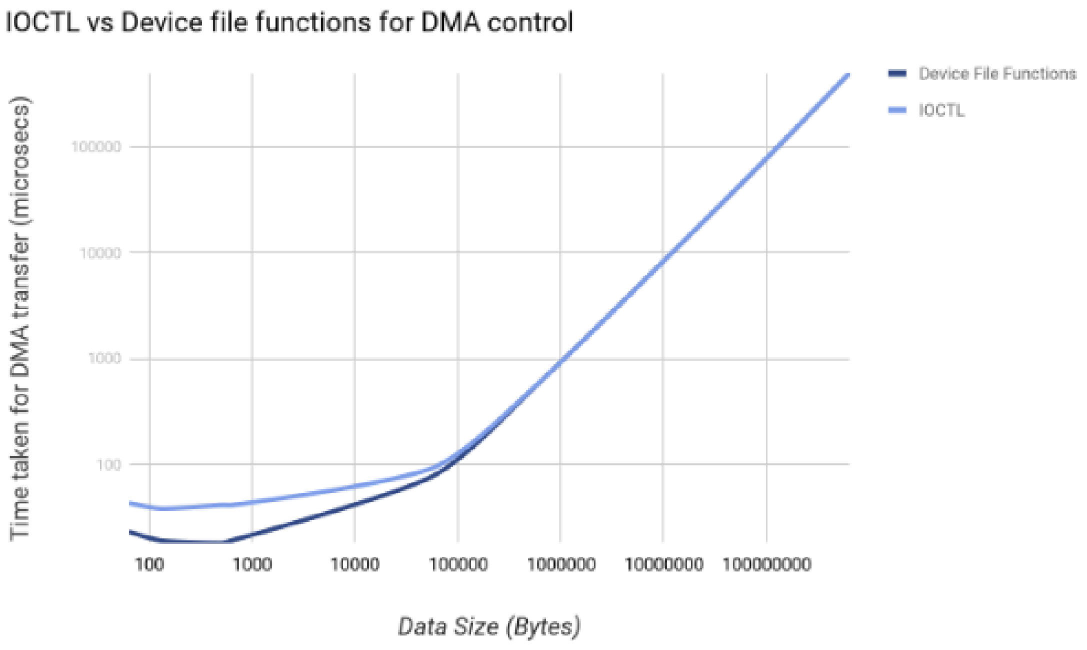

Figure 5.3: IOCTL vs Device File Function read and write rate

There are basically two ways to control the FPGA's read and write operations:

- IOCTL Function
- Device File inbuilt functions

The user application can use IOCTL functions to control the DMA transfer from the user space. In this method, a pointer is passed on to the kernel space which contains information about the source buffer size, offset and destination address. These address are received at the kernel space to control and trigger DMA transfer. The second and more broadly used mechanism is to use inbuilt device file functionalities. The user application can then just use these functions to transfer data. The implementation consist of both the methods of DMA transfer. Separate APIs are available to use either of the methods for DMA transfer. The test results show that the time taken for DMA transfer is significantly higher for smaller data sizes. But as the data size increases above 1MB, the transfer throughput is the same for both IOCTL and Device file read/write functions.

5.2.3 Comparison of Performance levels

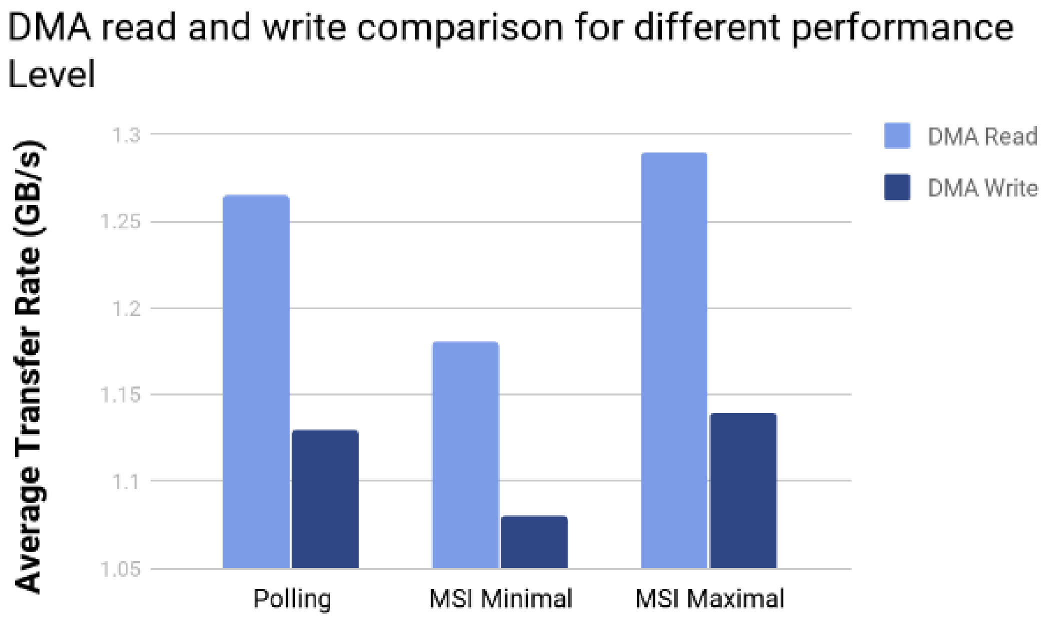

Figure 5.4: Comparison of Performance Levels

The three performance levels are compared and the average speed is taken for transfer of 640 MB of data. It can be seen that the throughput is fairly high for *Polling*. This is because of the efficient algorithm that employs parallel transfer of data from buffer user space to kernel space and from FPGA to kernel space. During polling, the status register is updated after every 1 MB transfer of data by the DMA descriptor. This immediate updating mechanism allows the next copy from user space to kernel space to take place immediately. In the case of MSI interrupts rather than polling, the driver needs to wait for all the descriptors to complete transaction for the interrupt to be triggered. Only when this interrupt is triggered will the kernel send the next copy from user space to kernel space. This increased performance of polling comes at a bigger cost - CPU clock cycles. Polling is a computer intensive task, as it needs to load the register continuously and compare if the status register has been updated. In the case of MSI interrupts, the thread goes to sleep until the interrupt is received. Therefore the CPU utilization will be much lower. The *MSI - Maximal Performance* is the best suited, as it overcomes the side-effects of polling with an increased throughput. However, it is to be noted that the the amount of buffers used by the *MSI - Maximal Performance* system is twice as that of *MSI - Minimal Performance*.

6 Summary and Future Work

By the end of the project, specification standards for PCIe, FPGA, Linux Centos 7 kernel have been studied and investigated in detail. A Linux management software was developed which would probe, enumerate and access the FPGAs connected to the host server through the PCIe. The driver creates device files and performs PIO read-write operations on the FPGA through device file functions. The driver would also perform DMA read and write operations from the user space to the FPGA and vice versa. Three levels of optimization were developed using MSI interrupts and polling. The performance of the three levels of optimization has been analyzed and depicted in the results section. The control of DMA read and write has been implemented through both ioctl and device file functions. The user has the option to choose between them. These three levels of optimization could be altered in real time by the user without creating any harm to the users currently using the system.

The driver has been developed such that multiple users and applications can simultaneously control and manage the FPGAs. A management layer parallelization was developed so that the user can manage and control multiple FPGAs simultaneously. The user has the option to choose the management layer as it comes with its own overhead. Separate user APIs have been developed to be used with and without the management layer for parallelization. A test environment has been developed for robust and versatile testing of the FPGA read-write operations. Separate test APIs have been developed for testing serial and parallel programming APIs based on the user's needs. A well-organized documentation has been produced to install, setup, test and use the driver and its APIs. In addition, an application has been developed to detect the number of FPGAs connected to the host server through PCIe. This application obtains the parameters of the running FPGAs including throughput, performance levels, temperature, etc to telegraf. Telegraf processes this information and sends them to a remote Database - InfluxDB. The users could access manage and view all these parameters of each and every FPGA connected through a user-friendly GUI from anywhere around the world.

Future work could include the following:

- Try to increase the number of descriptors for DMA transfer. The study has shown that the number of descriptors could increase the overall performance of DMA transfer.
- Partial Reconfiguration of the different FPGA sockets can be performed through PCIe. This would provide an added edge to the solution provider and increase the overall efficiency of the FPGA.

- In future, rather than connecting the FPGA through a PCIe to the server, it could be deployed as an individual entity that can be accessed through TCP or UDP packets.
- Tests could be conducted on different linux platforms to study the impact of the kernel and the hardware.

A User APIs

Note: Use -lpthread keyowrd during compilation of user application developed using the APIs. For example to compile exampleprogram.c using gcc, use command line:

gcc exampleprogram.c -o exampleprogram -lpthread

1.1. void getlasttransferrate (ssize_t file)
Prints the transfer rate of the FPGA Transfer rate includes read and write transfer rate. Read rate = transfer data rate from PC to FPGA. Write Rate = transfer data rate from FPGA to PC.

1.2. double getreadrate (ssize_t file)
Returns the data transfer rate from the host PC to the FPGA

1.3. double getwriterate (ssize_t file)
Returns the data transfer rate from FPGA to the host PC

1.4. ssize_t opendriver(int fpgacur)
Used to open the driver. Pass the ending digit of the FPGA to be opened. For example to open intelfpga1 device file, send 1 as argument. Returns 0 on failure. Returns file number on success. This file number should be used for all subsequent operations of the FPGA. The file number is used to identify the FPGA.

1.5. u_int64_t get_temperature(ssize_t file)
Returns Temperature of the Device

1.6. u_int64_t get_perflevel(ssize_t file)
Returns Performance Level of the Device

1.7. void performance_cntl(ssize_t file, int num_input)
Control the performance of the device. User can select which performance level the driver should work in by using the The 3 speeds specify the operating performance of the device driver.
0 - SPEED0 : Minimal Performance Â Uses Polling, 4 buffers of 1 MB each for read and write
1 - SPEED1 : Optimized Performance Uses MSI interrupts. 128 Buffers of 1MB each for read and write
2 - SPEED2 : High PErformance Uses MSI interrupts. 256 buffers of 1Mb each for read and write

This can be modified during runtime by user. Default performance is SPEED0

1.8. u_int64_t get_numofdevices(ssize_t file)
Returns Number of devices detected and enumerated by the driver.

1.9. void enable_msi(ssize_t file)
Enable MSI for DMA transfer. This API need not be used manually by the user as it is controlled by the 3 performance level. The user can just use the default performance levels to use msi or polling.This functionality is provided for developers who want to experiment further.

1.10. void enable_polling(ssize_t file)
Enable Polling for DMA transfer. This API need not be used manually by the user as it is controlled by the 3 performance level. The user can just use the default performance levels to use msi or polling. This functionality is provided for developers who want to experiment further

1.11. ssize_t fpgaread(ssize_t file, void *buf, size_t count, off_t offset)
PIO read. Used to transfer data from FPGA to user space in host PC. make sure the Â PIO is enabled before using enable_polling() function Arguments:
fd is the fpgadevice number to read
buf is the location pointer
count is the number of bytes to be read
offset is the offset location from which to be read
returns number of bytes read. returns -1 on failure;

1.12. ssize_t fpgawrite(ssize_t file, void *buf, size_t count, off_t offset)
PIO/DMA write. Used to transfer data from Host PC userspace to FPGA device. make sure the Â PIO is enabled before using enable_polling() function fd is the fpgadevice number to write
buf is the location pointer
count is the number of bytes to be written
offset is the offset location from which to be written
returns number of bytes written. returns -1 on failure;

1.13. void dmaread(ssize_t file, void Â *buf, u_int64_t len, u_int64_t off)
DMA read using ioctl functionality

1.14. void dmawrite(ssize_t file,void Â *buf, u_int64_t len, u_int64_t off)
DMA write using ioctl functionality

1.15. ssize_t fpgadmafromfpga(ssize_t file, void *buf, size_t count, off_t offset)
DMA read. Used to transfer data from FPGA to user space in host PC.
make sure the PIO is enabled before using enable_polling() function
fd is the fpgadevice number to read
buf is the location pointer
count is the number of bytes to be read
offset is the offset location from which to be read
returns number of bytes read. returns -1 on failure;

1.16. ssize_t fpgadmatofpga(ssize_t file, void *buf, size_t count, off_t offset)
DMA write. Used to transfer data from Host PC userspace to FPGA device. make sure the PIO is enabled before using enable_polling() function fd is the fpgadevice number to write
buf is the location pointer
count is the number of bytes to be written
offset is the offset location from which to be written
returns number of bytes written. returns -1 on failure;

1.17. void closedriver(ssize_t file)
Closes the driver file. Make sure to use this API on completion of application/usage of driver.

2. Test APIs
The following robust APIs are provided to give the developer ability to build the test environment custom needs

2.1 int pioaddresstest(ssize_t file)
PIO read test to check UUID It reads all the values of UUID and checks. The default UUID and LUID is present in this header file. Please make changes depending on the device. Returns 1 on success, 0 on failure.

2.2 int pioreadwritetest(ssize_t file)
PIO test to read and write into test registers. The location of the test register is given defined by TESTRGST_OFF in esg_fpga_pcie_userspace_api.h If successful returns 1 On failure returns 0;

2.3 int dmatest_diffoffset(ssize_t file, ssize_t rndlen, u_int64_t offset)
DMA test to read and write at multiple offset locations including different memory locations. returns -1 on if read or write is not possible, 0 on read write done but failed to be identical, 1 on successful completion

2.4. int fpgatest(ssize_t file)
Complete test of fpga Includes the following tests:
1. UUID read and checks if the IDs of all sockets match the requirement
2. Writes and reads random values to test registers of all the sockets. Verifies if the values written to test registers are identical.
3. Reads and then writes random data using DMA transfer and verifies if they are identical.
Returns 1 on success, 0 on failure

B Parallel Programming APIs

The following APIs are used for parallel programming and for usage and control of multiple FPGAs in parallel. This is an additional top layer used. The above functions should not be used in conjunction with the following APIs. Above functions return a file number on opening the driver file using opendriver() API. This file number is used for all subsequent actions including operation and control of the FPGA. The following layer returns a struct pointer variable struct 'fpgacontrol' (defined in esg_fpga_pcie_userpace_api.h) on opening the driver file using useropendriver() API. This pointer should be used for all subsequent actions and control of the FPGA.

```
struct fpgacontrol{
Â ssize_t file, len;
Â pthread_mutex_t lock;
Â u_int32_t inuseflag, testoption;
Â pthread_t userthread;
Â u_int64_t offset;
Â char Â *buf; u_int64_t off;
};
```

On using any of the following APIs, a new thread is called which performs the functions. Therefore the user can parallely do/work on other FPGAs on the main thread. This functionality can be used for any number of FPGAs. The variable inuseflag can be polled to check if the fpga has completed the work or if the fpga is currently in use.

IMPORTANT: Do not use the additional layer APIs in conjunction with the low level APIs above.

3.1. struct fpgacontrol* useropendriver(int fpganum)
Use this function to open a device driver. This opens up a driver and returns the fpga pointer which consist of various parameters like its lock, threadid, etc These parameters are not relevant to the layman user and is just used for high reliability and robustness. All subsequent read and writes should be made using the returned struct fpga. user should pass the fpganum that is the number of the device to be opened.

3.2. static int userfpgawrite(struct fpgacontrol *fpga, void Â *buf, u_int64_t len, u_int64_t off)

This function is used for user DMA write - transfer data from host PC user space to FPGA Pass the fpgacontrol struct that is returned on using fpgaopendriver() function. Other parameters that need to be passed includes:
buf: char pointer buf to which data needs to be written.
off: location from which data is to be read in fpga
len: length of the data to be written

3.3 static int userfpgaread(struct fpgacontrol *fpga, void Â *buf, u_int64_t len, u_int64_t off)
This function is used for user DMA read - transfer data from FPGA to host PC user space. Pass the fpgacontrol struct that is returned on using fpgaopendriver() function. Other parameters that need to be passed includes:
buf: char pointer buf from which data needs to be read.
off: location to which data is to be written in fpga
len: length of the data to be written

3.4 u_int64_t userget_temperature(struct fpgacontrol *fpga)
Returns Temperature of the Device

3.5 u_int64_t userget_perflevel(struct fpgacontrol *fpga)
Returns Performance Level of the Device. The 3 speeds specify the operating performance of the device driver.
0 - SPEED0 : Minimal Performance Â Uses Polling, 4 buffers of 1 MB each for read and write
1 - SPEED1 : Optimized Performance Uses MSI interrupts. 128 Buffers of 1MB each for read and write
2 - SPEED2 : High PErformance Uses MSI interrupts. 256 buffers of 1Mb each for read and write
This can be modified during runtime by user. Default performance is SPEED0 3.7 void userperformance_cntl(struct fpgacontrol *fpga, int num_input)
Control the performance of the device
0 = Polling
1 = MSI
2 = Enhanced MSI

3.6 u_int64_t userget_numofdevices(struct fpgacontrol *fpga)
Returns Number of devices detected and enumerated by the driver.

3.7 Â void userenable_msi(struct fpgacontrol *fpga)
Enable MSI for DMA transfer. This API need not be used manually by the user as it is controlled by the 3 performance level. The user can just use the default performance levels to use msi or polling. This functionality is provided for developers who want to experiment further

3.8 Â void userenable_polling(struct fpgacontrol *fpga)

Enable Polling for DMA transfer. This API need not be used manually by the user as it is controlled by the 3 performance level. The user can just use the default performance levels to use msi or polling. This functionality is provided for developers who want to experiment further

Parallel TEST APIs

4.1. static int userfpgatest(struct fpgacontrol *fpga, int option, ssize_t rndlen, long long int offset)

Used for testing the fpga. â€œOptionâ€ argument of function allows the user to test different types of testing based on the option parameter that is passed as argument.The options can be chosen as follow:

PIOADDRTST 110:

pioaddresstest()

Reads the UUID of each socket and checks if the value is the desired value as given by the UUIDH UUIDL

PIORDWRTST 111

pioreadwritetest()

Reads and writes multiple values to the fpga PIO test registers and checks if the values are correct.

BASICDMATST 112

basicdmatest()

Reads and writes random values into the FPGA and reads back and verifies if the read and write worked perfectly by comparing the written value to the read value.

FULLDMATST 113

fulldmatest()

Performs all the read and write DMA tests of the FPGA

FPGATST 114

fpgatest()

Performs all test of both the PIO and DMA read and write tests mentioned above.

IMPORTANT** the length and offset to be tested need to be passed only in case of basicdmatest

4.2. static int userbasicdmatest(struct fpgacontrol *fpga, ssize_t rndlen, long long int offset)

Used for direct DMA testing of FPGA without the need for using the â€œoptionâ€ argument as shown in the userfpgatest() API. Both userbasicdmatest() and with option BASICDMATST performs the same test function.

List of Figures

List of Tables

Bibliography

[1218] *Linux Device Drivers.* Accessed : 2018
`https://opensourceforu.com/tag/linux-device-drivers/`

[1718] *Nallatech 510T- Compute Acceleration Card Datasheet.* Accessed : 2018
`http://www.nallatech.com/wp-content/uploads/Nallatech-510T-Product-Brief-V2.2d.pdf`

[1910] *PCI Express Base Specification Revision 3.0.* November 10, 2010
`http://composter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf`

[2018] *Grafana Documentation 5.2.* Accessed : 2018
`http://docs.grafana.org/`

[2118] *InfluxDB 1.6 Documentation.* Accessed : 2018
`https://docs.influxdata.com/influxdb/v1.6/`

[2218] *Telegraf 1.8 documentation.* Accessed : 2018
`https://docs.influxdata.com/telegraf/v1.8/`

[5018] Linux Kernel Source Code, Accessed : 2018

[Abr15] Abraham, Kishon V.: PCI Express System Architecture, 1st Edition. In: *Overview of PCI(e) Subsystem, Linux Foundation Events*, 2015

[BB18] Blaise Barney, Lawrence Livermore National L.: POSIX Threads Programming, Accessed : 2018

[Ber96] Berg, Bil Lewis Daniel J.: PThreads Primer, A Guide to Multithreaded Programming, 1996

[BN11] Bradford Nichols, Jacqueline Farrell Jackie F. Dick Buttlar B. Dick Buttlar: PThreads Programming: A POSIX Standard for Better Multiprocessing, 2011

[Bot18] Bottomley, James: Linux Journal, Kernel Korner - Using DMA, Accessed : 2018

[Buc04] Buchanan, W. J.: "The Handbook of Data Communications and Networks", Springer Nature, 2004

[CPCAKMN18] CostaR. PaulA. ChakrabartiS. A. KhanJ. MitraT. Nayak, S. M.: An efficient approach to evaluate PCIe DMA design and DMA performance for Common Readout Unit(CRU), 24 February 2018

[Dow21] Downey, Allen B.: *The Little Book of Semaphores.* Version 2.2.1

[DSM18] David S. Miller, Jakub J. Richard Henderson H. Richard Henderson: The Linux Kernel documentation, Accessed : 2018

[IQPDS17] Intel Quartus Prime Design Suite: 18.0, Updated for: *Intel Arria 10 or Intel Cyclone 10 GX Avalon-MM DMA Interface for PCI Express* Solutions User Guide*, Nov 2017

[Jan15] Jangir, Mohn L.: *Linux Kernel and device driver programming : a simpler approach to Linux Kernel.* April 2015

[JC05] Jonathan Corbet, Greg Kroah-Hartman Alessandro R. Alessandro Rubini: Linux Device Drivers, 3rd Edition. In: *Linux Device Drivers, 3rd Edition*, 2005

[KB12] Kavianipour, H. ; Bohm, C.: High performance FPGA-based scatter/gather DMA interface for PCIe. In: *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*, 2012. – ISSN 1082–3654, S. 1517–1520

[KMB14] Kavianipour, H. ; Muschter, S. ; Bohm, C.: High Performance FPGA-Based DMA Interface for PCIe, 2014. – ISSN 0018–9499, S. 745–749

[KÅF$^+$18] Kekely, L. ; Å¡pinier, M. ; Friedl, Å¡. ; Å¡ikora, J. ; KoÅ™enek, J.: Live demonstration of FPGA based networking accelerator for 200 Gbps data transfers. In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018. – ISSN 2374–9709, S. 1–3

[Lae12] Laeeq, Shaikh M.: Mechanism of determining page faults instantaneously via device driver based approaches in Linux, IEEE CONFERENCE ON ELECTRICAL AND ELECTRONICS AND COMPUTER SCIENCE, 2012

[Lin12] Linux, Embedded: *Peter Barry, Patrick Crowley.* 2012

[Mad17] Madieu, John: *Linux Device Drivers Development.* October 2017

[PDL15] Paolo Durante, Rainer Schwemmer-Umberto Marconi Gabriele B. Niko Neufeld N. Niko Neufeld ; Lax, Ignazio: 100 Gbps PCI-Express Readout for the LHCb Upgrade, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 62, NO. 4, AUGUST 2015

[RB05] Ravi Budruk, Tom S. Don Anderson A. Don Anderson: PCI Express System Architecture, 1st Edition. In: *PCI Express System Architecture, 1st Edition*, 2005

[RCC+14] ROTA, L. ; CASELLE, M. ; CHILINGARYAN, S. ; KOPMANN, A. ; WEBER, M.: A new DMA PCIe architecture for Gigabyte data transmission. In: *2014 19th IEEE-NPSS Real Time Conference*, 2014, S. 1–2

[RG95] RAYMOND GREENLAW, WALTER L. R. H. JAMES HOOVER H. H. JAMES HOOVER: *Limits to Parallel Computation: P-Completeness Theory.* 1995

[SS17] SHANEHSAZZADEH, F. ; SADRI, M. S.: Area and performance evaluation of central DMA controller in Xilinx embedded FPGA designs. In: *2017 Iranian Conference on Electrical Engineering (ICEE)*, 2017, S. 546–550

[Ua17a] UG-01145$_avmm_dma$:
Arria 10 Avalon-MM DMA Interface for PCIe Solutions, *Dec*2017

[Ua17b] UG-01145$_avmm_dma$:
PCI Express AvalonÂ®-MM DMA Reference Design*, *Dec*2017

[Ven08] VENKATESWARAN, Sreekrishnan: *Essential Linux Device Drivers.* March 27, 2008

[WsCTS03] WEB SERVICES", Analysis "NAM: a network adaptable middleware to enhance response time o. 11th IEEE/ACM International Symposium on Modeling M. 11th IEEE/ACM International Symposium on Modeling ; COMPUTER TELECOMMUNICATIONS SYSTEMS, Simulation of: *S. Ghandeharizadeh, C. Papadopoulos, P. Pol, R. Zhou.* 2003

[XL17] XIAOXIAO LI, Jingguo Ge Hongbo Zheng Yuepeng E Chunjing Han Honglei L. Yulei Wu W. Yulei Wu: "A kernel-space POF virtual switch", Computers Electrical Engineering, 2017

[Zha10] ZHANG, Peng: "Industrial control system operation routines", Advanced Industrial Control Technology,, 2010

[ZLBAM15] ZAZO, J. F. ; LOPEZ-BUEDO, S. ; AUDZEVICH, Y. ; MOORE, A. W.: A PCIe DMA engine to support the virtualization of 40 Gbps FPGA-accelerated network appliances. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015, S. 1–6