RDMA over Converged Ethernet

Sherif Badran, Rami Rasheedi, Zeyad Madbouly, et al.
Computer and Systems Department
Faculty of Engineering, Ain Shams University
Cairo, Egypt

July 25, 2021

Ain Shams University, Faculty of Engineering

Electronics and Communications Engineering Department

Cairo, Egypt

RDMA over Converged Ethernet

A Report Submitted in Partial Fulfillment of the Requirements of the Degree of Bachelor of Science in Electronics and Communications Engineering

By

Mohammed Hussien Mostafa	**1601160**
Mohamed khaled Mohamed Sayed El khawas	**1601173**
Christine Magdy Gad El-Rab Samuel	**16E0129**
Martina Fadi Fouad farag	**1601053**
Maryham melad Gerges Michael	**1601075**

Supervised by

Dr. Ashraf Salem

Cairo 2021

Declaration

We hereby certify that this project submitted as part of our partial fulfillment of BSc in Electronics and Communications Engineering is entirely our own work, that we have exercised reasonable care to ensure its originality, and does not to the best of our knowledge breach any copyrighted materials, and have not been taken from the work of others and to the extent that such work has been cited and acknowledged within the text of our work.

Signed

Name	Signature
Mohammed Hussien Mostafa	محمد حسين
Mohamed khaled Mohamed Sayed El khawas	محمد الخواص
Christine Magdy Gad El-Rab Samuel	
Martina Fadi Fouad farag	
Maryham melad Gerges Michael	

Date: Day of 24th of July, in the year 2021.

Acknowledgments

Thank and acknowledge your advisor, family and friends.

Abstract

Nowadays, data have become a crucial aspect of all evolving computer technologies. That's why owning data and knowing how to use it efficiently has become a key of success to any enterprise. Not only is data a crucial aspect of our modern life, but also the speed at which applications are running and data can be accessed has a significant impact as well. However, massive amounts of data that need to be analyzed, processed, shared, transferred, monitored and accessed (Big data/parallel computing/Cloud computing) have led to very high work load on data centers and systems. This has led to slower processing speeds and increased latency in response to user requests or operations running. In addition, the high availability requirement of any database has become exceedingly important and backup systems are taken into consideration from the very beginning at the phase of designing the system to ensure no or minimum down time and continuous functionality.

Our project proposes a methodology to tackle the above problems where applications can directly access the memory and perform (I/O) operations without (CPU) interference through (DMA) directly. Hence, minimizing latency and maximizing the (CPU) processing speed.

(RDMA) allows direct access of memory of one computer into the other's memory without involving any OS. This is very useful nowadays in the distributed systems where individual computers are connected together and are communicating together easily to facilitate efficient data transfer and parallel processing and resource sharing to appear as one integrated system.

There are various (RDMA) protocols such as (RoCE V.1) and (RoCE V.2) and (IWARP). Ethernet is an alternative (RDMA) offering that is more complex and unable to achieve the same level of performance as RoCE-based solutions.

However, in our project, we have implemented (RoCE V.2). This is an internet protocol which allows accessing memory between different hosts within different domains directly over an Ethernet network. The details of our version and implementation will be discussed later on.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

Abbreviations	
API	Apllication Programming Interface
BAR	Base Address Register
CPU	Central Processing Unit
CQ	Completion Queue
CQE	Completion Queue Element
CQE	Completion Queue Element\{}\{}
CRC	Cyclic Redundancy Checks
DDR	Double Data Rate
DID	The Device ID registers
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FCFS	First come, First served
FCS	Frame Check Sequence
FIFO	First-in First-out
FSM	Finite State Machine
HCA	Host Channel Adapter
HCA	Host Channel Adapter\{}\{}
HPC	High Performance Computing
HWM	High watermark
I/O	Input / Output
IB	InfiniBand
IBTA	InfiniBand Trade Association
IETF	Internet Engineering Task Force
iWARP	Internet wide-Area Network
LWM	Low watermark
MTU	Maximum transmission unit
NIC	Network Interface Card
OS	Operating System

PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
QP	Queue Pair
RDMA	Remote Direct Memory Access
RLDP	Row Locality based Drain Plicy
RoCE	RDMA over Converged Ethernet
RoCE V.1	RDMA over Converged Ethernet version 1
RoCE V.2	Routable RoCE
SSID	The Subsystem ID
SVID	the Subsystem Vendor ID
TCP	Transmission Control Protocol
TLPs	Transaction Layer Packets
VID	Vendor ID registers
WOE	Work Queue Element
WQE	Work Queue Element\{}\{}

Chapter 1

Introduction

1.1 Project Proposal

Our aim was to design a high performance programmable memory controller that utilizes external (DRAM) memories to be shared remotely over network (I/O) without consuming the system memory resources and with minimal host (CPU) intervention. This proposal offers a plug-and-play approach for (RoCE) systems that can extend the capabilities of a conventional hardware architecture to be used in Distributed Memory Systems.

(RDMA) provides direct access from the memory of one computer to the memory of another through network (I/O) without involving either computer's operating system. This technology enables high-throughput, low-latency networking with low (CPU) utilization, which is especially useful in massively parallel compute clusters. That's why (RDMA) is quickly becoming a necessity in performance-critical networking nowadays since most of the applications require intensive processing and data analysis.

Our project's edge lies in the fact that we are in an era where owning data and knowing how to use it efficiently has become the key to success for any enterprise. Therefore, companies are competing for owning larger amounts of significant data from people to extract useful information that can help their businesses improve based on the data they obtained. So we can say that data now has become as equally valuable as money and profits for companies.

Consequently, new fields of science have emerged to help in dealing with the huge amounts of data flooding over the internet and various applications. Data scientists, data analysts and data engineers are now concerned with dealing, storing and extracting useful information from all kinds of data and storing them in datacenters and data lakes for future use.

1.2 Project Overview

Due to the numerous benefits that have been discussed earlier regarding the (RDMA), we decided to implement it. Currently, there are various (RDMA) protocols such as: (RoCE) and (RoCE V.2) and (iWARP). In our project we have implemented (RoCE V.2) or in other words (RoCE) and the choice of this version specifically that we have implemented will be later discussed in details.

RoCE is a network protocol defined in (IBTA) standard, allowing (RDMA) over converged Ethernet network. Shortly, it can be regarded as the application of (RDMA) technology in hyper-converged data centers, cloud, storage, and virtualized environments. It possesses all the benefits of (RDMA) technology and the familiarity of Ethernet. (RDMA) over Converged Ethernet (RoCE) is the most commonly used (RDMA) technology for Ethernet networks and is deployed at scale in some of the largest "hyper-scale" data centers in the world.

In traditional sockets networks, applications request network resources from the operating system through an (API) which conducts the transaction on their behalf. However (RDMA) use the OS to establish a channel and then allows applications to directly exchange messages without further (OS) intervention. A message can be an (RDMA) Read or Write, (RDMA) Send or Receive operation.

(RDMA) provides low latency through stack bypass and copy avoidance, reduces (CPU) utilization, reduces memory bandwidth bottlenecks and provides high bandwidth utilization. The key benefits that (RDMA) delivers accrue from the way that the (RDMA) messaging service is presented to the application and the underlying technologies used to transport and deliver those messages. (RDMA) provides Channel based (I/O). This channel allows an application using an (RDMA) device to directly read and write remote virtual memory. (RoCE) bypasses the (CPU) allowing data to be transferred between hosts directly through Ethernet connection between them.

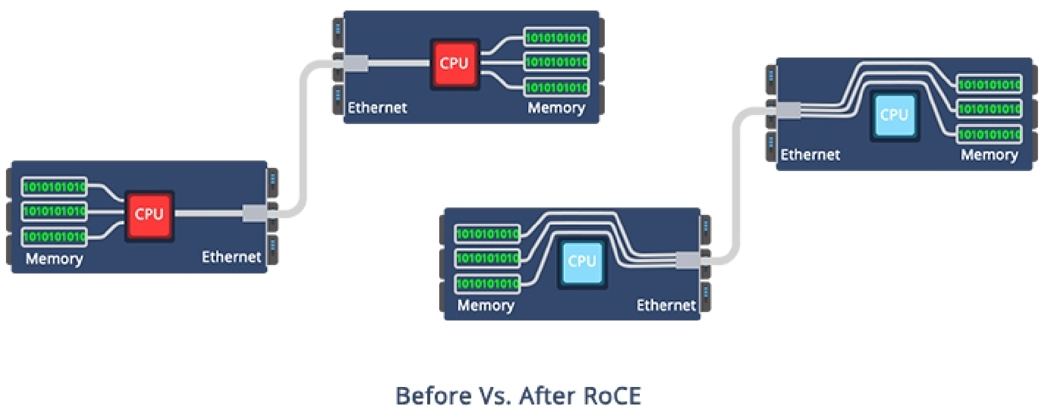


Figure 1.1: Before VS After RoCE

Every application has direct access to the virtual memory of devices. This means that applications do not need to make requests to an operating system to transfer messages as with the traditional network environment where the shared network resources are owned by the operating system and cannot be accessed by a user application. Thus, an application must rely on the involvement of the operating system to move data from the application's virtual buffer space, through the network stack and out onto the wire.

In the coming figures, we can observe the difference between traditional network topology and when (RDMA) is implemented. Each application requesting to access another application's memory will issue a request to be transferred through the network interface controller passing by the operating system of the other host.

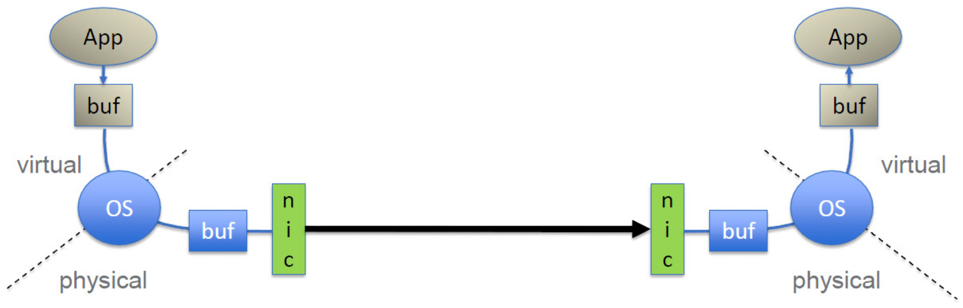


Figure 1.2: Traditional memory access topology

In our proposal, applications can directly access each other's memory with no operating system intervention. Thus, minimizing CPU utilization in memory read/writes and minimizing CPU delay imposed by these operations and increasing the efficiency required to perform more sophisticated operations.

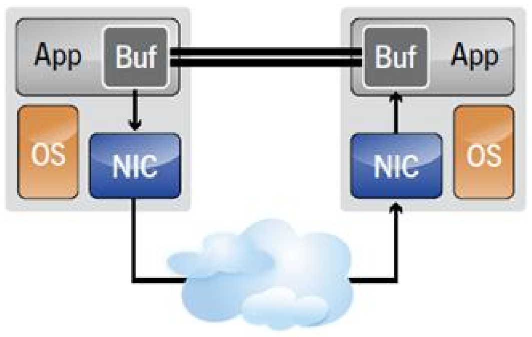


Figure 1.3: (RDMA) memory access topology

1.3 RDMA Protocols

As we have mentioned earlier, (RDMA) protocols which are (RoCE V.1) and RoCE V.2 (Routable RoCE) and (iWARP).

1.3.1 iWARP

The (iWARP) protocol defines how to perform (RDMA) over a connection-oriented transport like the Transmission Control Protocol (TCP). The memory requirements of a large number of connections along with (TCP)'s flow and reliability controls lead to scalability and performance issues when using (iWARP) in large-scale datacenters and for large-scale applications. Furthermore, multicast is defined in the RoCE specification while the current (iWARP) specification does not define how to perform multicast (RDMA). This protocol is supported by some vendors such as Cheliso and Intel.

1.3.2 RoCE

It is a network protocol that allows remote direct memory access (RDMA) over an Ethernet network. It is the most commonly used (RDMA) technology for Ethernet networks and is deployed at scale in some of the largest "hyper-scale" data centers in the world. The emerging (RDMA) over Converged Ethernet (RoCE) standards enables the (IB) transport for use over the existing and widely deployed Ethernet infrastructure. It does this by encapsulating an (IB) transport packet over Ethernet. InfiniBand (IB) is a computer networking communications standard that allows high throughput (processing rates) and low delays used when data is transferred between computers or between servers in datacenters.

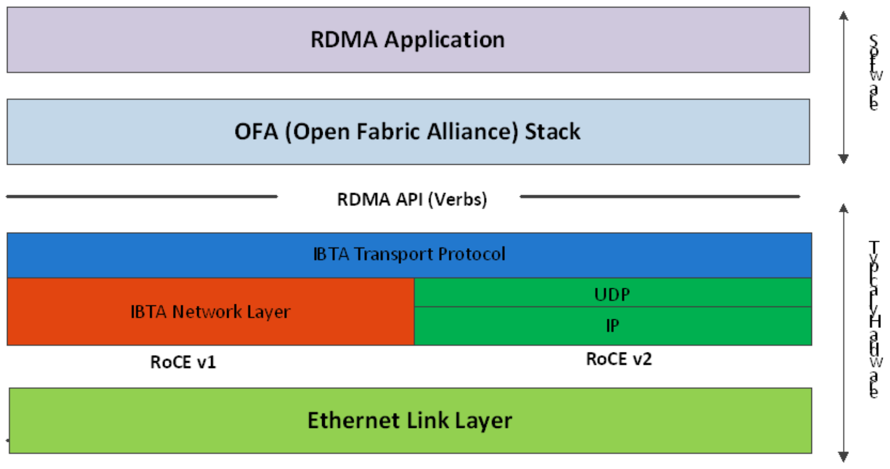


Figure 1.4: RDMA Architecture Stack

1.3.3 RoCE v1 vs. RoCE v2

(RoCE V.1) is an Ethernet link layer protocol. This means that the frame length limits of the Ethernet protocol apply: 1500 bytes for a regular Ethernet frame and 9000 bytes for a jumbo frame. It allows communication between any two hosts in the same Ethernet broadcast domain. (RoCE V.1) is limited to a single Ethernet broadcast domain. The Ethernet broadcast domain (VLAN) is the domain where nodes reach each other though broadcasts at datalink layer.

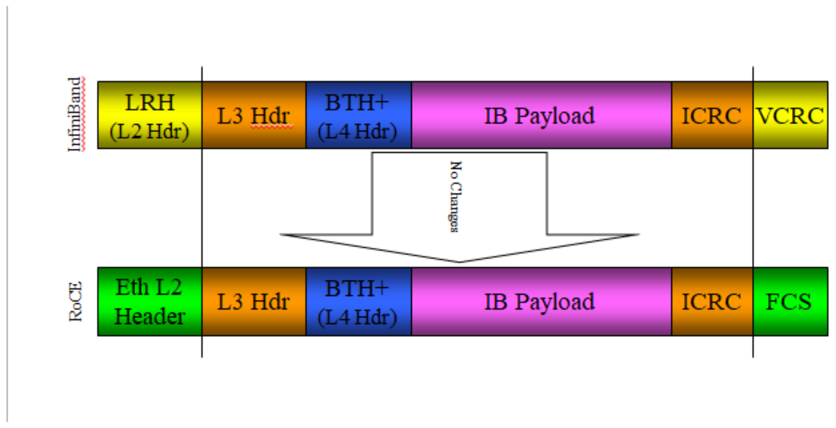


Figure 1.5: (RoCE V.1) Packet Format

(RoCE V.2) is an internet layer protocol since it connects multiple networks through gateways and enables communication across IP subnets. It is responsible for transporting and routing packets from host to destination specified by the IP addresses. It exists on top of either the UDP/IPv4 or the UDP/IPv6 protocol. Since (RoCE V.2) packets are routable the (RoCE V.2) protocol is sometimes called Routable RoCE.

In addition, (RoCE V.2) defines congestion control mechanism to deal with congestion that may occur during packets transmission. Software support for (RoCE V.2) is still emerging, that's why our project is concerned with implementing this version of the (RDMA) protocol.

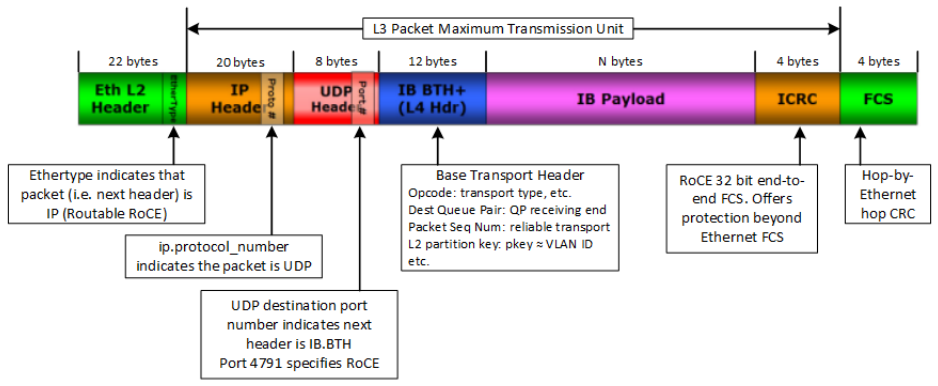

L3 Packet Maximum Transmission Unit
22 bytes
20 bytes
8 bytes
12 bytes
N bytes
4 bytes
4 bytes
Eth L2 Header
EtherType
IP Header
Proto #
UDP Header
Port #
IB BTH+ (L4 Hdr)
IB Payload
ICRC
FCS
Ethertype indicates that packet (i.e. next header) is IP (Routable RoCE)
ip.protocol_number indicates the packet is UDP
UDP destination port number indicates next header is IB.BTH Port 4791 specifies RoCE
Base Transport Header
Opcode: transport type, etc.
Dest Queue Pair: QP receiving end
Packet Seq Num: reliable transport
L2 partition key: pkey ≈ VLAN ID
etc.
RoCE 32 bit end-to-end FCS. Offers protection beyond Ethernet FCS
Hop-by-Ethernet hop CRC

Figure 1.6: RoCE v2 packet format

1.3.3.1 FCS

The frame check sequence (FCS) is an extra field in each transmitted frame that can be analyzed to determine if errors have occurred. The (FCS) uses (CRCs), checksums, and two-dimensional parity bits to detect errors in the transmitted frames.

1.3.3.2 BTH

The Opcode field identifies the type of request or response packet.The DestQP field identifies the destination QP within the remote CA. The destination QP is responsible for handling the incoming packet. The PSN field contains the packet's 24-bit Packet Sequence Number. The

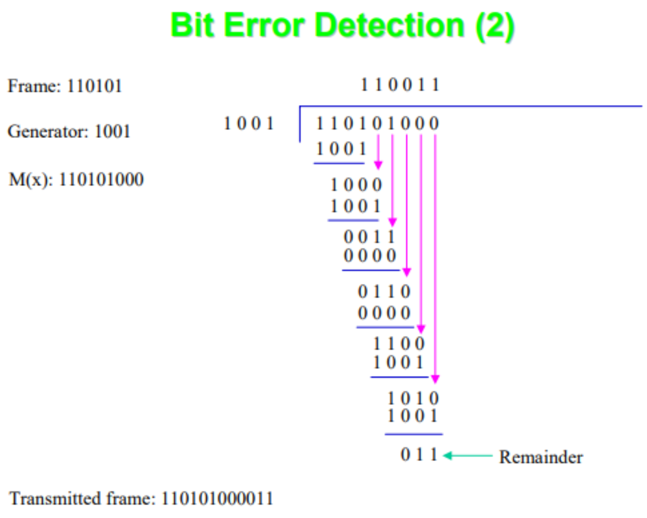


Figure 1.7: Error Detection

PSN is inserted into a transfer request packet by the initiator and is checked for correctness by the recipient. This permits the recipient to detect missing packets.

1.4 How packets are built

A packet is built starting with the application layer data, and headers from protocols operating on lower layers are added as the packet is being built, moving from top to bottom. This means that the last protocol header that is added is at the Data Link layer, which means that we should encounter this header first. The most common data link layer protocol is Ethernet.

An Ethernet frame is preceded by a preamble and start frame delimiter (SFD), which are both part of the Ethernet packet at the physical layer. Each Ethernet frame starts with an Ethernet header, which contains destination and source MAC addresses as its first two fields.

The frame size of a standard Ethernet frame (defined by RFC 894) is the sum of the Ethernet header (14 bytes), the payload (IP packet, usually 1,500 bytes), and the Frame Check Sequence (FCS) field (4 bytes). However, a jumbo frame, with an (MTU) size of 9,000 bytes, can also be configured.

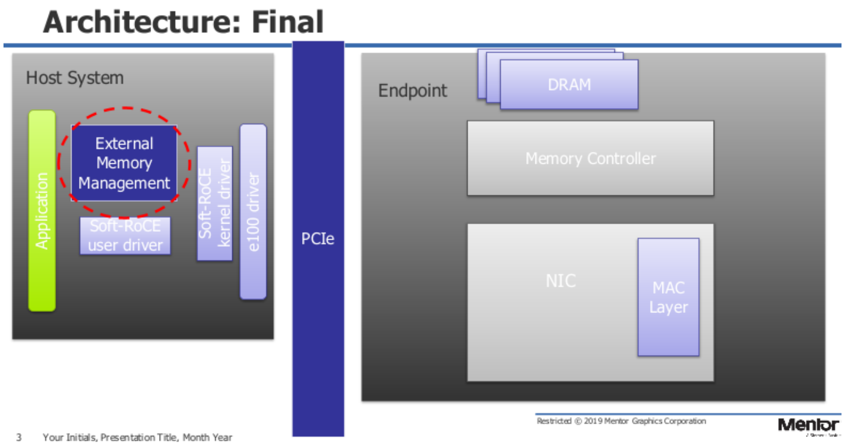


Figure 1.8: Overall Architecture

1.5 Project Architecture

Our project is mainly divided into two components integrated together: the host system and the (PCIe) endpoint. The Vlab pcie host solution inetgrates both sides together. It integrates the rtl code used to implement the memory controller with the software side represented through the host system (Qemu Virtual Lab Solution) to verify the endpoint connection.

- The host system was launched on the Virtual Lab Solution Qemu. A simple (RDMA) application is setup there to test and show the (RDMA) connectivity between server and client hosts. Then we have the Soft-RoCE user and kernel drivers.
- The drivers' main functionality is to interface with the memory controller through the (PCIe) bus providing (API)s needed to configure the (PCIe) endpoint memory and perform memory read and write operations while offloading the (CPU) and ensuring no operating system intervention.
- For this reason, we have implemented our own memory driver with the functions required to interface with the memory controller which we have implemented as well to enable the user to configure and use the external dynamic ram (DRAM) of the (PCIe) endpoint directly without passing by the system kernel over network (I/O). This memory driver is responsible of dealing with memory read/write requests to be performed on the (PCIe) endpoint (chip connected through RTL emulation).
- RoCE drivers are different from the conventional known driver. Traditionally, when a user desires to perform an operation that can be done only through kernel, user drivers interface with kernel drivers who accomplish the required task. However, in (RoCE), we have used certain intermediate libraries (libibverbs and uverbs) that act as an intermediate layers between user-level and kernel-level drivers to enable the user to access the memory directly bypassing the kernel drivers required before. Therefore, user level drivers now have access to memory and address space directly.
- E100 driver is the network driver that is responsible of handling the Ethernet traffic packets passing through the network interface card (NIC) and checking the packets' headers. We chose this network driver because it is a simple driver and we were able to go through it and understand how it works.
- The memory controller which has local buffer storage and special purpose registers. The controller is responsible of reading and writing the data between the (PCIe) endpoint device memory and the local buffer storage of our controller. Hence, ensuring speedy data transmission and minimal (CPU) utilization required in the data transfer process.

Algorithm 1.1 Anything

```
Require: $\rho \geq 1$
Ensure: $X_k$
1: while not converged do
2:     Solve $X_{k+1} = \min_X L(X, Y_k, \mu_k)$
3:     $Y_{k+1} = Y_k + \mu_k h(X_{k+1})$
4:     $\mu_{k+1} = \rho\mu_k$
5: end while
```

Chapter 2

RDMA Architecture

2.1 How (RDMA) works

In (RDMA) we setup data channels using a kernel driver. We call this the command channel. We use the command channel to establish data channels which will allow us to move data bypassing the kernel entirely. Once we have established these data channels we can read and write buffers directly.

(RDMA) operations start by “pinning” memory. When you pin memory, you are telling the kernel that this memory is owned by the application. Now we tell the (HCA) to address the memory and prepare a channel from the card to the memory. We refer to this as registering a Memory Region.

2.1.1 Queue Pairs

(RDMA) communication is based on a set of three queues. The send queue and receive queue are responsible for scheduling work. They are always created in pairs. A Completion Queue (CQ) is used to notify us when the instructions placed on the work queues have been completed.

2.1.2 Work Queue Elements (WQE)

A user places instructions on its work queues that tell the (HCA) what buffers it wants to send or receive. These instructions are small structures called work requests or Work Queue Elements (WQE). A (WQE) placed on the send queue contains a pointer to the message to be sent. A pointer in the (WQE) on the receive queue contains a pointer to a buffer where an incoming message from the wire can be placed. When the WQE is processed the data is moved. Once the transaction completes a Completion Queue Element (CQE) is created and placed on the Completion Queue (CQ). We call a (CQE) a “COOKIE”.

2.1.3 How the sequence is done

1. After the two communicating systems have created their QP’s Completion Queue’s and registered regions in memory for (RDMA) to take place. First system identifies a buffer that it will want to move to the second system. The second system has an empty buffer allocated for the data to be placed.

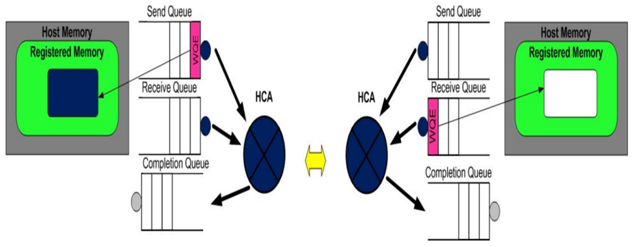


Figure 2.1: CQ creation

2. Second system creates a WQE "WOOKIE" and places in on the Receive Queue. This WQE contains a pointer to the memory buffer where the data will be placed. The first system also creates a WQE which points to the buffer in its memory that will be transmitted.

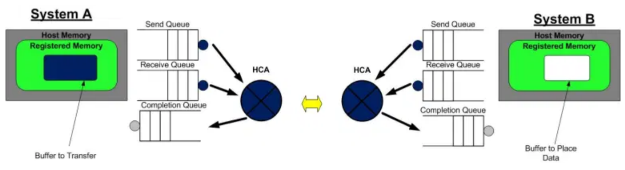


Figure 2.2: WQE creation

3. The (HCA) is always working in hardware looking for WQE's on the send queue. The (HCA) will consume the (WQE) from the first system and begin streaming the data from the memory region to second system. When data begins arriving at System B the (HCA) will consume the (WQE) in the receive queue to learn where it should place the data. The data streams over a high-speed channel bypassing the kernel.

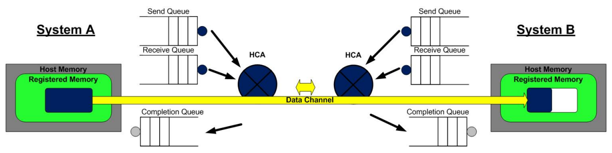
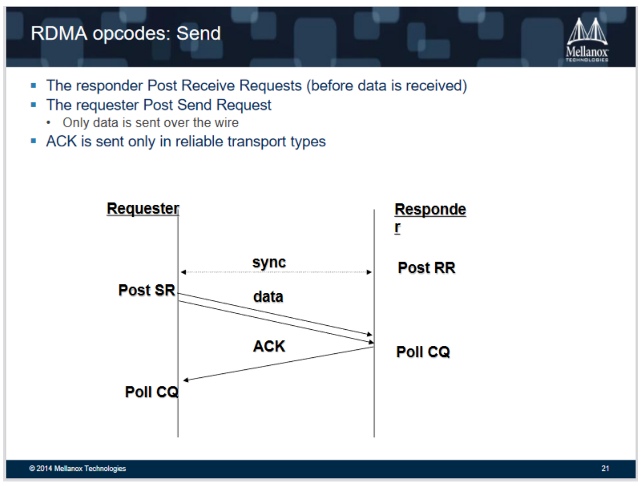


Figure 2.3: Data Channel Creation

4. When the data movement completes the (HCA) will create a (CQE) "COOKIE". This is placed in the Completion Queue. For every (WQE) consumed a (CQE) is generated. First system's (CQ) indicating that the operation completed and also on second system's CQ. A CQE is always generated even if there was an error. The (CQE) will contain field indicating the status of the transaction.

2.2 RDMA Operations

- RDMA SEND: The send operation allows sending data to a remote QP's receive queue. The receiver must have previously posted a receive request and acquire receive buffer to receive the data. The sender does not have any control over where the data will reside in the remote host.



RDMA opcodes: Send

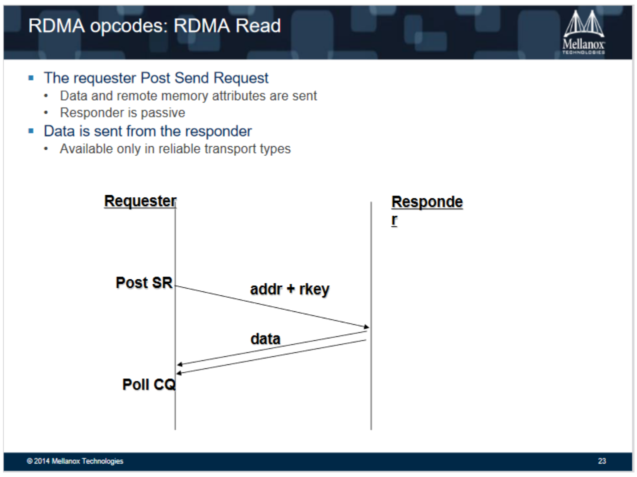
- The responder Post Receive Requests (before data is received)
- The requester Post Send Request
  - Only data is sent over the wire
- ACK is sent only in reliable transport types

Requester
Responde r
sync
Post RR
Post SR
data
ACK
Poll CQ
Poll CQ

© 2014 Mellanox Technologies 21

Figure 2.4: RDMA Send

- RDMA READ: A section of memory is read from the remote host. The caller specifies the remote virtual address as well as a local memory address to be copied to. Prior to performing (RDMA) operations, the remote host must provide appropriate permissions to access its memory. Once these permissions are set, (RDMA) read operations are conducted with no notification whatsoever to the remote host. For both (RDMA) read and write, the remote side isn't aware that this operation being done (other than the preparation of the permissions and resources). This is due to no CPU involvement throughout the entire process.

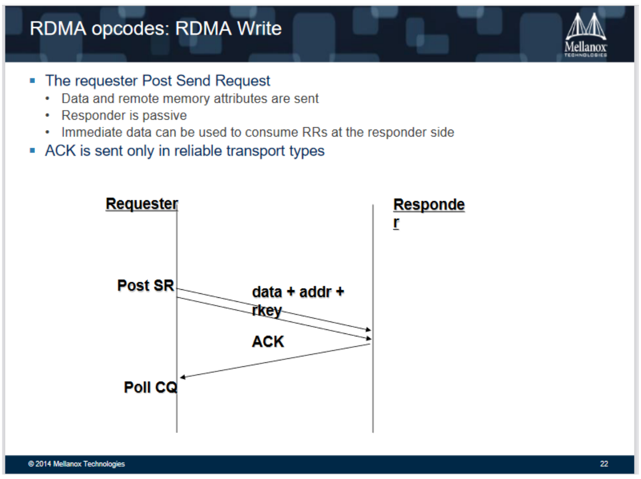


RDMA opcodes: RDMA Read

- The requester Post Send Request
  - Data and remote memory attributes are sent
  - Responder is passive
- Data is sent from the responder
  - Available only in reliable transport types

Figure 2.5: RDMA Read

- RDMA WRITE: Similar to (RDMA) read, but the data is written to the remote host. (RDMA) write operations are performed with no notification to the remote host Remote keys are given to the remote (HCA) to allow a remote process access to system memory during (RDMA) operations.



RDMA opcodes: RDMA Write

- The requester Post Send Request
  - Data and remote memory attributes are sent
  - Responder is passive
  - Immediate data can be used to consume RRs at the responder side
- ACK is sent only in reliable transport types

Figure 2.6: RDMA Write

2.3 RDMA Transport Modes

RDMA supports various transport modes such as:

1. RC: reliable connection where packets are delivered in order and each (QP) is associated with one (QP) fulfilling reliable delivery. Queue Pair is associated with only one other (QP). Messages transmitted by the send queue of one (QP) are reliably delivered to receive queue of the other (QP). Packets are delivered in order. (RC) connection is very similar to a (TCP) connection. This is the transport mode that we are using in our (RDMA) application.

2. UC: unreliable connection as packets may be lost.

3. UD: unreliable datagram where one (QP) may send or receive from any other UD (QP) since no actual connection is formed between queue pairs. Packets aren't ordered during delivery and maybe lost. This mode is similar to UDP connection.

4. RD: reliable datagram [not supported by API library].

Chapter 3

Client server flow

3.1 Protocol

There are many ways we could orchestrate the transfer of an entire file from client to server.
For instance:

- Load the entire file into client memory, connect to the server, wait for the server to post a set of receives, then issue a send operation (on the client side) to copy the contents to the server.
- Load the entire file into client memory, register the memory, pass the region details to the server, let it issue an (RDMA) read to copy the entire file into its memory, then write the contents to disk.
- As above, but issue an (RDMA) write to copy the file contents into server memory, and then signal it to write to disk.
- Open the file on the client, read one chunk, wait for the server to post a receive, then post a send operation on the client side, and loop until the entire file is sent.
- As above, but use (RDMA) reads.
- As above, but use(RDMA) writes.

But loading the entire file into memory can be impractical for large files, so we will skip the first three options. Of the remaining three, we will focus on using (RDMA) writes so that we can illustrate the use of the (RDMA)-write- with immediate-data operation.

Now we decided that we are going to break up the file into chunks, and write the chunks one at a time into the server's memory, we must find a way to ensure that we do not write chunks faster than the server can process them. We will do this by instructing the server to send explicit messages to the client when it is ready to receive data. The client, on the other hand, will use writes with immediate data to signal the server. The sequence looks something like this:

3.2 Communication sequence

1. Server starts listening for connections.
2. Client posts a receive operation for a flow-control message and initiates a connection to the server.
3. Server posts a receive operation for an (RDMA) write with immediate data and accepts the connection from the client.
4. Server sends the client its target memory region details.
5. Client re-posts a receive operation then responds by writing the name of the file to the server's memory region. The immediate data field contains the length of the file name.

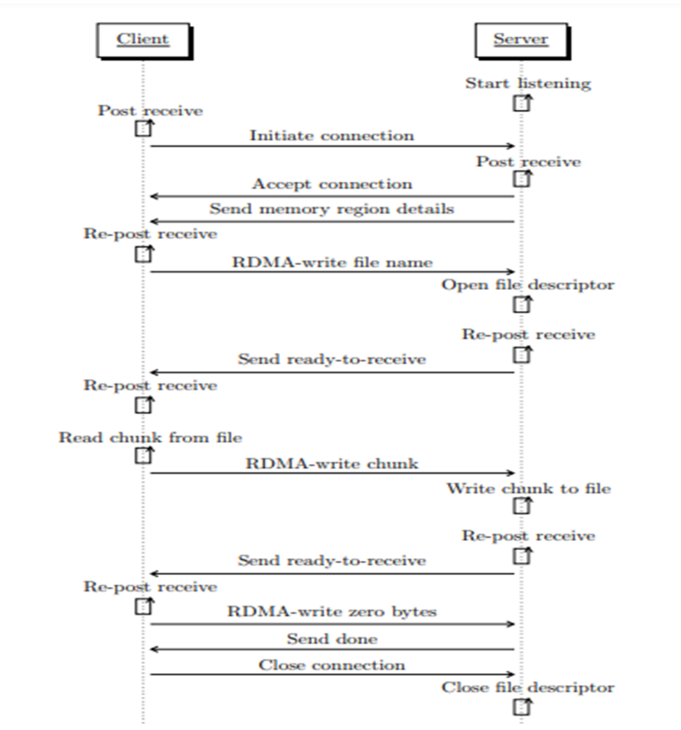


Figure 3.1: Client server sequence diagram

6. Server opens a file descriptor, re-posts a receive operation, then responds with a message indicating that it is ready to receive data.
7. Client re-posts a receive operation, reads a chunk from the input file, then writes the chunk to the server's memory region. The immediate data field contains the size of the chunk in bytes.
8. Server writes the chunk to disk, re-posts a receive operation, then responds with a message indicating that it is ready to receive data.
9. Repeat steps 7 and 8 until there are no data left to send.
10. Client re-posts a receive operation, and then initiates a zero-byte write to the server's memory. The immediate data field is set to zero.
11. Server responds with a message indicating that it is finished.
12. Client closes the connection.
13. Server closes the file descriptor.

3.2.1 The program flow on Server Side

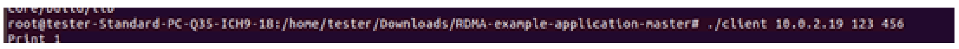
1. Create an event channel.
2. Bind to an address.
3. Create a listener to listen at the port for connection request.
4. Create a protection domain, completion queue, and send-receive queue pair.
5. Post-receive before you accept the connection.
6. Accept the connection request.
7. Wait for the receive completion.
8. Send results back to client.

3.2.2 The program flow on Client Side

1. Create an event channel.
2. Resolve the peer's address.
3. Resolve the route to the peer.
4. Create a protection domain, completion queue, and send-receive queue pair.
5. Connect to server.
6. Wait for the connection to be established.
7. Pre-post receives.
8. Send data to the server.
9. Wait for receive completion and receive reply.

3.3 Application Description

Coming to the soft (RoCE)application, we started by setting up soft (RoCE) on QEMU host system. This phase required intensive understanding of linux commands to be able to build the (RoCE) module and install the required libraries in the process. After this, we started executing the (RDMA) application. This application represents a simple addition operation to be done on to numbers sent from the client side to be added at the server side and return back the summation result.



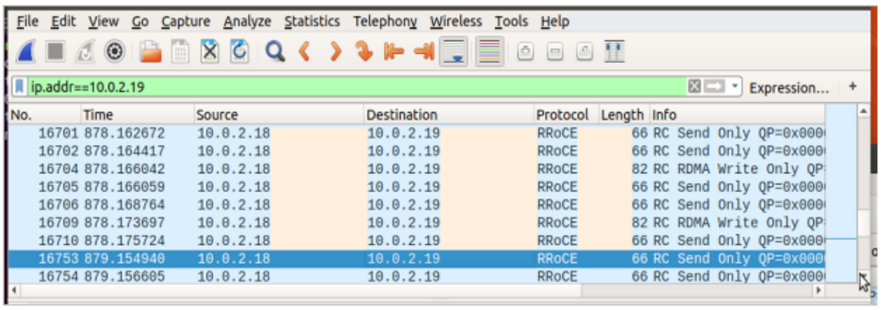
```
root@tester-Standard-PC-Q35-ICH9-18:/home/tester/Downloads/RDMA-example-application-master# ./client 10.0.2.19 123 456
Print 1
```

Figure 3.2: Snapshot of client.c execution

This snapshot shows the execution of client code using " ./client " command and specifying the ip address of the server "10.0.2.19" then it sends the two numbers "123"&"456" that are required to be added using the server.

3.4 Wireshark observations

We used wireshark to observe the traffic packets sent and received on the channel between server and client.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

ip.addr==10.0.2.19

No.	Time	Source	Destination	Protocol	Length	Info
16701	878.162672	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16702	878.164417	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16704	878.166042	10.0.2.18	10.0.2.19	RRoCE	82	RC RDMA Write Only QP
16705	878.166059	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16706	878.168764	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16709	878.173697	10.0.2.18	10.0.2.19	RRoCE	82	RC RDMA Write Only QP
16710	878.175724	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16753	879.154940	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000
16754	879.156605	10.0.2.18	10.0.2.19	RRoCE	66	RC Send Only QP=0x000

Figure 3.3: Wireshark execution snapshot

This snapshot shows an example of the send (RDMA) write operation involving the (QP) address of each host captured from wireshark while application execution.

3.4.1 Sent packets

The sequence starts by sending “connect request” from the client to the server asking for connection with the server then the server replies by “connect reply” &”class port info” back to the client , at this moment client will send “ready to use packet” announcing that it’s ready to send or receive data packets to and from the server.

3.5 Application APIs

3.5.1 Client Operation

A general connection flow follows:

1. rdma_create_event_channel
   - Creates a channel to receive events.
2. rdma_create_id
   - Allocates an rdma_cm_id identifier that is conceptually similar to a socket.
3. rdma_resolve_addr
   - Obtains a local Remote Direct memory Access (RDMA) device to reach the remote address.
4. rdma_get_cm_event
   - Waits for the RDMA_CM_EVENT_ADDR_RESOLVED event.
5. rdma_ack_cm_event
   - Acknowledges the received event.
6. rdma_create_qp
   - Allocates a queue pair (QP) for the communication.
7. rdma_resolve_route
   - Determines the route to the remote address.
8. rdma_get_cm_event
   - Waits for the RDMA_CM_EVENT_ROUTE_RESOLVED event.
9. rdma_ack_cm_event
   - Acknowledges the received event.
10. rdma_connect
    - Connects to the remote server.
11. rdma_get_cm_event
    - Waits for the RDMA_CM_EVENT_ESTABLISHED event.
12. rdma_ack_cm_event
    - Acknowledges the received event.
13. ibv_post_send()
    - Performs data transfer over the connection.
14. rdma_disconnect

- Tears down the connection.

15. rdma_get_cm_event
    - Waits for the RDMA_CM_EVENT_DISCONNECTED event. rdma_ack_cm_event Acknowledges the event.
16. rdma_destroy_qp
    - Destroys the (QP).
17. rdma_destroy_id
    - Releases the rdma_cm_id identifier.
18. rdma_destroy_event_channel
    - Releases the event channel.

Note: In the example, the client initiated the disconnect. However, either the client or server operation can initiate the disconnect process.

3.5.2 Server operation

A general connection flow follows:

1. rdma_create_event_channel
    - Creates a channel to receive events.
2. rdma_create_id
    - Allocates an rdma_cm_id identifier that is conceptually similar to a socket.
3. rdma_bind_addr
    - Sets the local port number on which the event listens.
4. rdma_listen
    - Starts listening to the connection requests.
5. rdma_get_cm_event
    - Waits for the RDMA_CM_EVENT_CONNECT_REQUEST event with a new rdma_cm_id identifier.
6. rdma_create_qp
    - Allocates a queue pair (QP) for the communication on the new rdma_cm_id identifier.
7. rdma_accept
    - Accepts the connection request. rdma_ack_cm_event Acknowledges the event.
8. rdma_get_cm_event
    - Waits for the RDMA_CM_EVENT_ESTABLISHED event.
9. rdma_ack_cm_event
    - Acknowledges the event. ibv_post_send() Performs the data transfer over the connection. rdma_get_cm_event Waits for the RDMA_CM_EVENT_DISCONNECTED event.
10. rdma_ack_cm_event Acknowledges the event.
    - rdma_disconnect Tears down the connection. rdma_destroy_qp Destroys the(QP).

11. rdma_destroy_id
    - Releases the connected rdma_cm_id identifier.
12. rdma_destroy_event_channel
    - Releases the event channel.

Chapter 4

PCIe End Point Memory Management

4.1 Modules

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module.

4.1.1 Character Devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. Char devices that look like data areas, and we can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

4.1.2 Block devices

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In Linux Systems, it allows the application to read and write a block device like a char device, it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Block drivers have a completely different interface to the kernel than char drivers.

4.1.3 Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

4.2 A Driver

This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Driver is a program that interacts with a particular device or special kind of software. It contains special knowledge of the device or special software interface that programs using the driver do not. The Linux way of looking at devices distinguishes between the predefined three fundamental device types.

4.2.1 A Driver Purpose

The rate at which new hardware becomes available alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets.

We are able to make our own choices about the driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is “flexible,” we like this word because it emphasizes that the role of a device driver is providing mechanism, not policy. The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: “what capabilities are to be provided” (the mechanism) and “how those capabilities can be used” (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

Our focus was on a network driver, that is a software program that controls a device used to connect a host to a network. Network drivers control the interface between a host and a given network. It is familiar with the protocol being used by the host network, creating a unique identification for the host that can be used in the system. As information is exchanged between host and the network, the network driver converts it into usable formats. The network driver also provides feedback to the user about the status of the network so that people know at all times when they are connected, at what speed, and if there are any problems with the network.

Hosts can have one or more networking devices, including wireless cards or wired ethernet cards, for example. Without network drivers, these devices cannot work properly and may have trouble accessing the network or executing commands from the user.

4.2.2 The objective of especially using E100 Ethernet Driver

E100 driver is a network driver that supports 10/100 Mbps PCI Ethernet controller that are used in client/server network interface cards. This driver accesses share memory and controls status registers (CSR) of devices. CSR registers are responsible of set up, configuration, queuing of TX and RX commands. It is the driver that is responsible of handling the Ethernet traffic packets passing through the network interface card (NIC) and checking the packets’ headers. We chose this network driver because it is a simple driver and we were able to go through it and understand how it works. It was written by the same company that designed and manufactured the device, so it knows how to communicate with the device hardware to get the data. Consequently, It is a software component that lets the linux kernel and (PCIe) endpoint communicate with each other. Hence,

- If an application needs to read some data from a device, the application calls a function implemented by the linux operating system.
- The operating system calls a function implemented by the driver.
- And the driver calls a function implemented by device controller by loading the appropriate registers within the device controller.
- Then, the device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”).

- After that, the controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation.
- The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read, which returns it to the application.
- For other operations, the device driver returns status information.

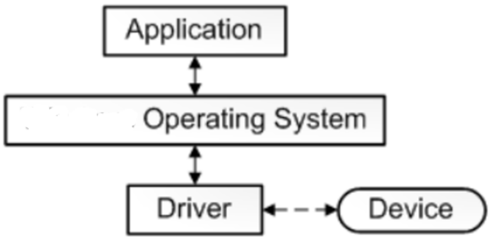


Figure 4.1: Host System

- To know more about device controller, It is in charge of a specific type of device. Depending on it, more than one device may be attached. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) controller. It maintains some local buffer storage and a set of special-purpose registers. And it's responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Typically, operating systems have a device driver for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

For Example:

```
e1000 code snippet:
MODULE_DEVICE_TABLE(pci, e1000_pci_tbl);
MODULE_DEVICE_TABLE(pci, foo_ids);
static struct pci_driver foo_driver = {
.name = e1000e,
.id_table = e1000_pci_tbl,
.probe = e1000e_probe,
.remove = e1000e_remove
};
```

4.3 Linux Kernel Role

A Driver is required for the communication between the kernel and any connected (PCIe) end-point to turn the controller's operations and requests into the Kernel's generic APIs. Therefore, (CPU) resources can be allocated and system-memory accessed independent on the type of end-point connected. So, we should know how the kernel's role can be split into the following different parts, and how to use the PCIe endpoint:

4.3.1 Kernel

is the core part of Linux, It is responsible for all major activities of this operating system, It consists of various modules. It provides the required abstraction to hide low level hardware details to system or application programs. It interacts directly with hardware, provides low level services to upper layer components. It is part of the multitasking system responsible for the management of memory communication between them.

In a Linux system, several concurrent processes attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other

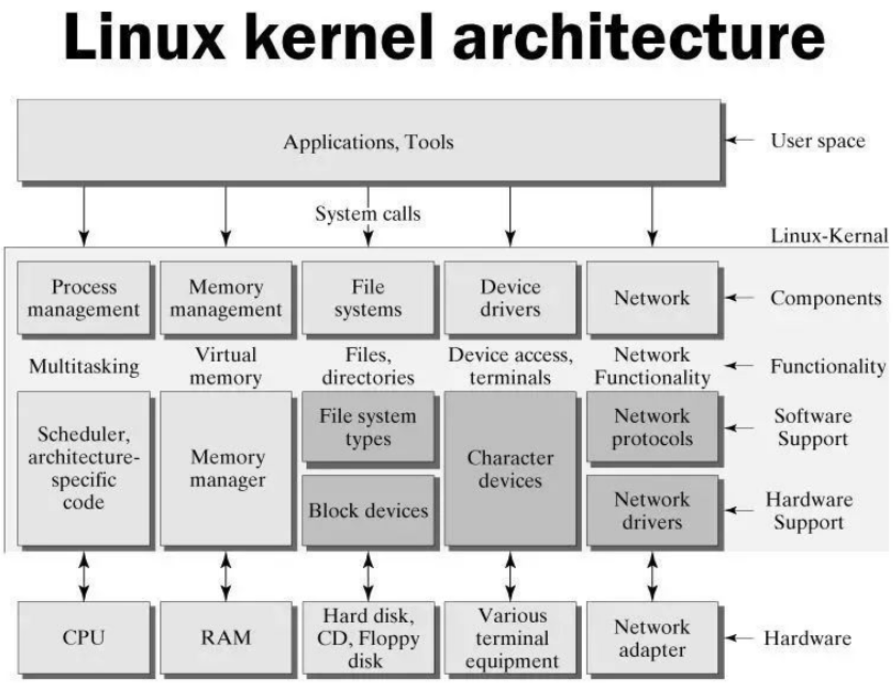


Figure 4.2: Linux Architecture

resource. The kernel is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked.

4.3.2 Memory management

The host's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.

4.3.3 Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the (CPU), is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single (CPU) or a few of them.

4.3.4 Filesystems

Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system.

4.3.5 Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

4.3.6 Networking

It must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel. Therefore, Our focus was on kernel memory management. As mentioned earlier, the project essential goal was to design a high performance programmable memory controller that utilizes external DRAM memories to be shared remotely over network (I/O) without consuming the system memory resources and with minimal host (CPU) intervention. And (RDMA) that provides direct access from the memory of one host to the memory of another through network (I/O) without involving either host's operating system, enables high-throughput, low-latency networking with low (CPU) utilization, which is especially useful in massively parallel compute clusters.

4.4 PCIe Endpoint

4.4.1 Its Characteristics

An endpoint is a remote computing device that communicates back and forth with a network to which it is connected. It terminates a (PCIe) link; it only has one connection to the (PCIe) tree topology, it can have connection to another kind of bus. It can also act as a bridge to legacy/compatibility bus, such as a PCIe-to-PCI bridge. It's a device that resides at the bottom of the branches of the tree topology and implements a single Upstream Port toward the Root. Native PCIe Endpoints are (PCIe) devices designed from scratch. Endpoint may support (IO) transactions, and may support locked transaction semantics as a completer but not as a requester. Endpoints are always device 0 on a bus. Multi-Function Endpoints. Like (PCI)devices, (PCI) Express devices may support up to 8 functions per endpoint with at least one function being number 0.

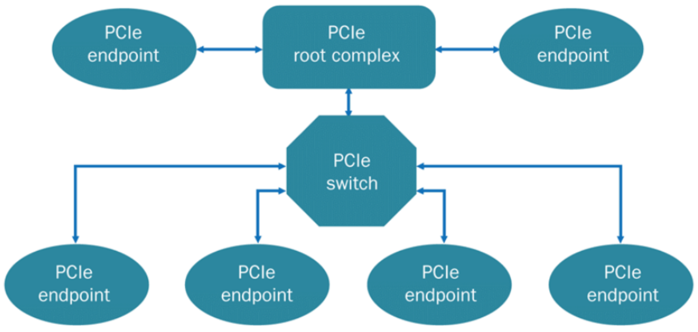


Figure 4.3: Pcie tree topology

Therefore, We have used the (PCIe) endpoint in our project as a chip connected to (PCIe) bus emulated through Virtual lab Solution.

4.4.2 PCIe Endpoint Detection

The probe function is called by the (PCI) core when it has a struct pci_dev that it thinks the driver wants to control. If the driver claims the struct pci_dev that is passed to it. It initializes the device by registering facilities used by the module and tells the kernel that it is a net_device. It then enables the (PCI) bus and sets up the related memory regions and registers.

4.4.3 Configuration Space

It is typically 256 bytes, and can be accessed with Read/Write Configuration Cycles

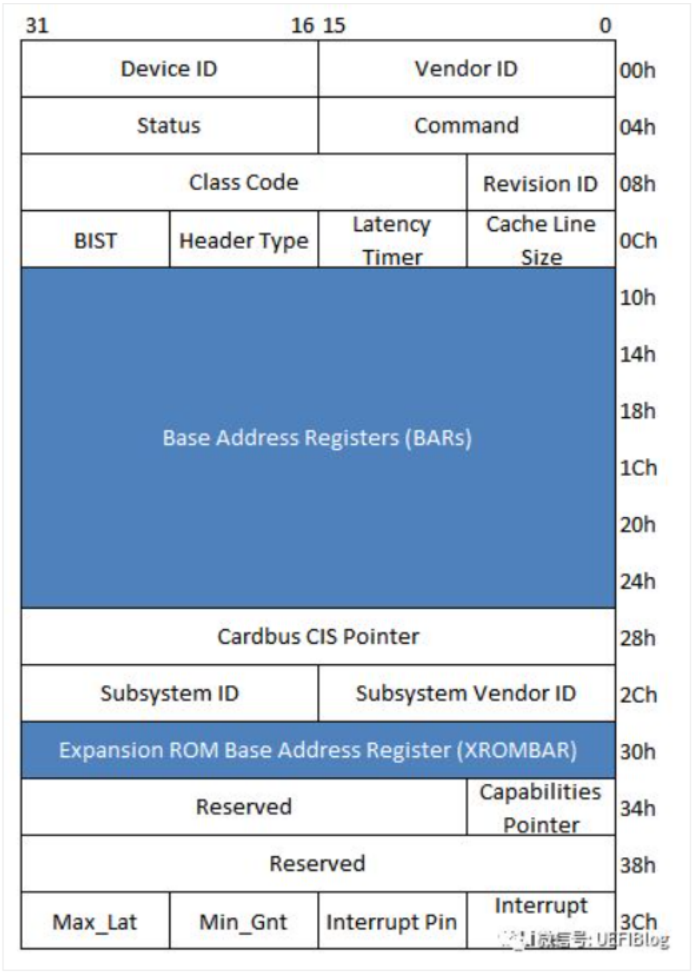


Figure 4.4: pcie endpoint configuration space

1. The Device ID (DID) and Vendor ID (VID) registers: They identify the device (such as an IC), and are commonly called the (PCI) ID. The 16-bit vendor ID is allocated by the PCI-SIG. The 16-bit device ID is then assigned by the vendor. There is an inactive project to collect all known Vendor and Device IDs.

2. The Status register: It is used to report which features are supported and whether certain kinds of errors have occurred.

3. The Command register: It contains a bitmask of features that can be individually enabled and disabled.

4. The Header Type register: Its values determine the different layouts of remaining 48 bytes (64-16) of the header, depending on the function of the device. That is, Type 1 headers for Root Complex, switches, and bridges. Then Type 0 for endpoints.

5. The Cache Line Size register: It must be programmed before the device is told it can use the memory-write-and-invalidate transaction. This should normally match the CPU's cache line size, but the correct setting is system dependent. This register does not apply to PCI Express.

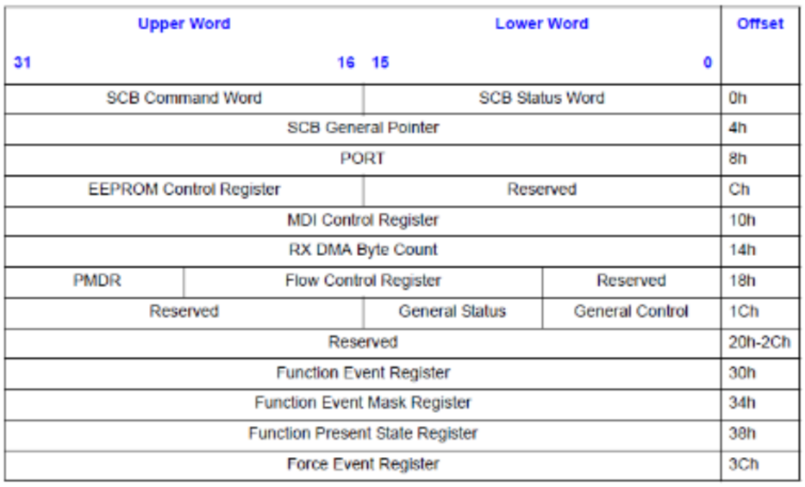
6. The Subsystem ID (SSID) and the Subsystem Vendor ID (SVID): They differentiate specific model (such as an add-in card). While the Vendor ID is that of the chipset manufacturer,

the Subsystem Vendor ID is that of the card manufacturer. The Subsystem ID is assigned by the subsystem vendor from the same number space as the Device ID. Generally, the Vendor ID–Device ID combination designates which driver the host should load in order to handle the device, as all cards with the same VID:DID combination can be handled by the same driver. The Subsystem Vendor ID–Subsystem ID combination identifies the card, which is the kind of information the driver may use to apply a minor card-specific change in its operation.

7. The Base Address Register: • BARs are the starting address of a contiguous mapped address in system memory or I/O space. In PCIe, an Endpoint requests a size of contiguous memory (or I/O space), which is then mapped by the upstream device's memory manager, and the Base Address Register is programmed with the base address for that Endpoint's BARx field in the endpoint's configuration space. They are filled by Linux Kernel.For example, a 32-bit BAR0 is offset 10h in PCI Compatible Configuration Space , and post enumeration would contain the start address of BAR0.
8. They represent memory windows as seen by the host system (CPUs) to talk to the device. The device doesn't write into that window but answers (TLPs) requests (MRd*, MWr*).
9. So, (BAR) is basically the device's way to tell the host how much memory it needs, and of what type. For each (BAR), the system will read the size of the required memory window and allocate a physical address range for access to that window. On the user side, we typically get signals indicating which BAR was active (which address window was used for this transaction). we then need to decode this information to decide what to do with the transaction.

4.4.4 PCIe Endpoint Register Space

The endpoint communicates with the driver through a set of special purpose registers that reside on the PCIe endpoint side.



Upper Word		Lower Word		Offset
31	16	15	0	
SCB Command Word		SCB Status Word		0h
SCB General Pointer				4h
PORT				8h
EEPROM Control Register		Reserved		Ch
MDI Control Register				10h
RX DMA Byte Count				14h
PMDR	Flow Control Register		Reserved	18h
Reserved		General Status	General Control	1Ch
Reserved				20h-2Ch
Function Event Register				30h
Function Event Mask Register				34h
Function Present State Register				38h
Force Event Register				3Ch

Figure 4.5: pcie endpoint register space

4.5 The goal of Memory Offloading

There are a number of reasons to be interested in the memory offloading which means replacing the system memory with PCIe endpoint memory to save system resources, and register PCIe endpoint memory.

- RoCE drivers are different from the conventional known driver. Traditionally, when a user desires to perform an operation that can be done only through kernel, user drivers interface with kernel drivers who accomplish the required task. However, in RoCE, we have used

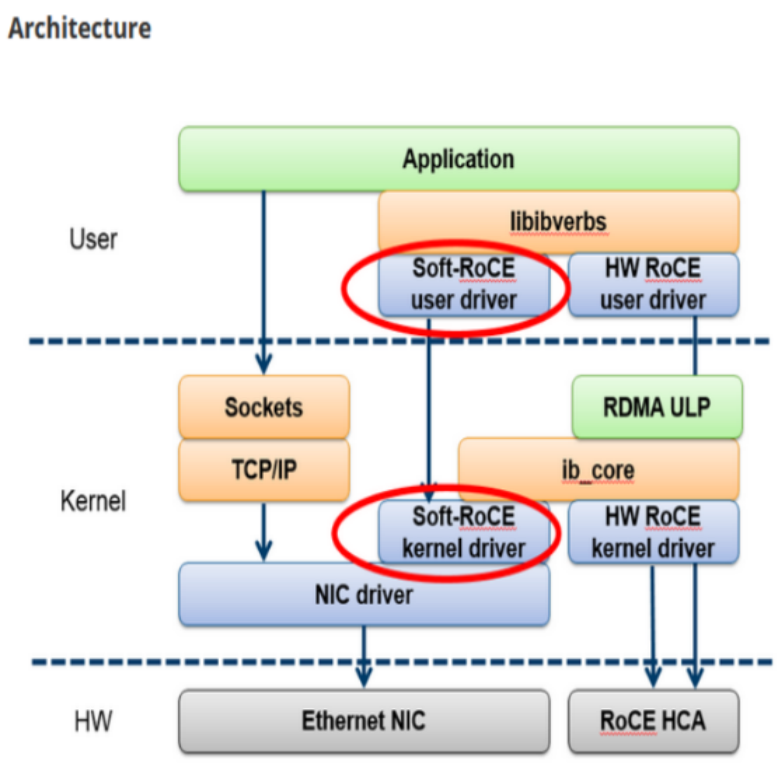


Figure 4.6: Soft RoCE Stack

certain intermediate libraries (libibverbs and uverbs) that act as an intermediate layers between user-level and kernel-level drivers to enable the user to access the memory directly bypassing the kernel drivers required before. Therefore, user level drivers now have access to memory and address space directly.

- The aim to have a vendor specific user-level driver is that verbs exposes directly the HCA's hardware registers to the userspace and each (HCA) has a different set of registers, then the need for an intermediate userspace layer. The fact that RDMA devices can translate virtual addresses on their own means that the userspace library does not have to go through the kernel in order to obtain the physical address of the buffer when creating entries in the work queues.

- These ibverbs and uverbs libraries implement the user-kernel communication. Some operations cannot be done entirely in userspace, e.g. registering memory or opening/closing device contexts, and those are transparently passed to the kernel module by libibverbs. uverbs module maintains idr tables that are used to translate between kernel pointers and opaque userspace handles. Kernel can then keep track of which resources are attached to a given userspace context.This also allows the kernel to clean up when a process exits and prevent one process from touching another process's resources.

- Also, we should realize that the necessity of memory registration is that the PCIe device allows the host to access a small sized memory block of the device's memory, which prevents the host from accessing the rest of the device's memory. And It is often desirable for the host to access the full memory space of an endpoint device for debugging, configuration or other purposes, while avoiding reserving a large memory space in the host memory, like Allocate memory, Read memory, Write memory, Mapping memory (indicate the regions of memory or memory mapped IO to the endpoint via configuration space (BAR) registers), and all (DMA) operations functions.

Chapter 5

Introduction to Memory systems

Chapter 6

The DDR5 Technology

To make sure we fully understand the memory controller we will review the ddr5 dram system.

1. The memory consists of no.of bank groups in each bank group there is no.of banks which is the grid of cells containing the data.
2. Every command between the controller and the request must be in bursts (16 column next to each other in the same row).
3. Every columns have a storage for 16 bits.
4. There is a small timing constraints between sending commands to banks in different bank groups.
5. There is a medium timing constraints between commands to different banks in the same bank group.
6. There is a large timing constraints between sending two commands in the same bank.
7. There is timing constraints between sending write command to the memory and sending read command to it.
8. When we want to read data from memory:
   (a) We must send activate command to the the row containing the required data to be fetched to enable the transistors to connect the data in the capacitor to flow to the sense amplifier.
   (b) 8.2. We must send the burst address to be fetched then the data burst will come in order so you can take the columns you need.
   (c) 8.3. If we need to issue other command to other row in the same bank we will issue precharge command first to store the values from the sense amplifiers to the capacitors because the read and write operations are destructive to the data in the capacitors.
9. When we want to write to the memory:
   (a) We follow the read procedure but we make sure to choose the data mask right to make sure that the data stored in the columns that we don't want to change will remain the same.
10. There is time we must wait between every to commands that vary based on the time the other commands issued on to the memory.
11. With time passing the capacitor charge is changing due to that the transistor is not perfect isolator so every while the memory needs to be refreshed by reading all the stored data in the capacitors to the sense amplifier then writing it again to it's capacitors.

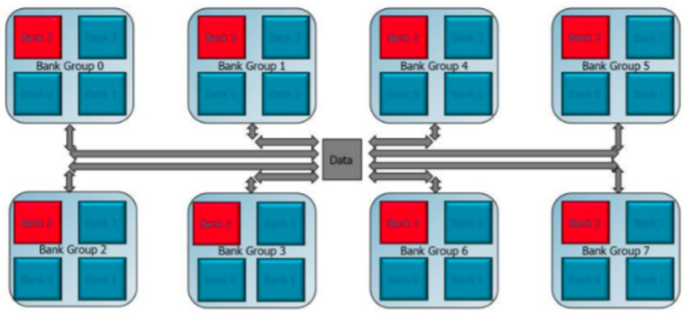


Figure 6.1: DDR5 Bank Groups

6.1 Row policy

There are three types of the row management police:

1. Open page policy

   (a) After issuing the read or write command we don't precharge the active row until we find a row conflict in bank then we precharge the row.

   (b) It saves a lot of operation if there is a lot of requests in the same row follows each other then a lot of operation costed us time.

   (c) Uses a lot of power in case that the row still active and the traffic is small then the row will be active for long time with no commands issued to that bank.

   (d) If the requests mapping scheme is bad then there will be a huge loss of time waiting for the next request to come then finding out it's not the active row so then we will start to precharge the row.

2. Closed page policy

   (a) After issuing the read or write command we issue precharge command immediately to the active row in the bank.

   (b) It saves a lot of time if there is a lot of requests in different rows in the same bank follows each other.

   (c) Uses less power in case that the traffic is small in comparison to the open row policy 2.4. If the requests mapping scheme is bad then it will be better than the open page policy because some of the time between the precharge command activate command will pass before the next burst come.

3. Hybrid

   (a) It acquires the advantage in the time optimization and the power usage from the two policies.

   (b) It needs a lot of designing effort and time to make it fit to the requests traffic behaviour.

   (c) It's complex to design and adds a lot of hardware size to the chip.

So we finally decided to choose the open page policy because our expected requests comes close to each other and in the same row as much as possible according to the mapping scheme and our hardware space and development time is limited.

Chapter 7

DDR5 Controller Architecture

7.1 Front-end

7.1.1 Transaction controller

It's the controller interface with the user module by the mapper and the returner modules and it has an interface with the modified fifo by the request saver module .

7.1.1.1 Mapper

Applies mapping scheme and gives every request special index.

1. It takes the input address, data and request type from the user if the valid port is active.
2. then it calculates where to place it inside the memory according to the mapping scheme to make sure the data fetching is as fast as possible.
3. then it add a unique index to every request inside the memory controller to make sure the returner can but the requests the right order.
4. then send it to the request saver without the (bank or bank group) bits to lower the overall controller internal memory storage.
5. And it activate the busy port if there is no storage left in the controller or no unique indices to be added to the new requests.

7.1.1.2 Returner

1. It has a counter for the read index and other for the write index which start with zero and it's incremented if the request withe the same index enter the returner.

2. It waits for specific request index to come.

3. If any other request index came it store it inside an internal storage until it's index become the required request index to be sent to the user.

4. If the request index came and it was a read request it sends the data to the user with read_done flag set as high and increment the read index counter.

5. If it was write request it only sets the write_done flag to high and increment the write index counter.

7.1.1.3 Overflow stopper

It make sure that there are no two requests in the memory controller having the same index as each other By sending stop_reading or stop_writing signals to the mapper to prevent it from issuing new index to other requests and telling the user stop sending requests by activating the busy port until there is new indices available (until request is done from the memory with the same type as the indices we don't have)

7.1.1.4 Request saver

it make sure to store the request sent by the mapper in case of the target bank modified FIFO's storage if the FIFO has empty place to store the request then it forward the request if the FIFO is full it will store the request in internal storage until the bank FIFO is ready to receive the request

7.1.2 Modified FIFO (per bank)

It has two FIFOs buffer inside one for the write request's data which has low entries number and another for the common parts between the read and write request (type, address and index) with high entries number so it can store the read and write requests in order without increasing the internal storage by making the all the stored request with the data bits It allows the read request to use the empty write buffer instead of making all of the memory controller stops receiving requests due to lack of internal storage It saves and assign the request's data to it's FIFO buffer if it's write request only

7.1.3 Bank scheduler (per bank)

7.2 Back-end

7.2.1 Arbiter

7.2.2 Timing controller

2.2 timing controller: 1. it stores the active row address in each bank if it exists to decided the command to send if there is burst to send in that bank and it updates the active rows addresses on issuing the activate and precharge commands 2. Then it takes the state of each burst storage in the burst handler and it's bank, bank group and row address 3. Then if there is requests to be sent stored in any burst storage it start checking the necessary commands to be sent for each bust storage 4. If any command is sent it make it's counter zero then on the next time there is other command ready to be sent it checks the commands counters to make sure there is no timing constraints violation before sending the next command 5. Then it chooses which one of the ready bursts commands to be sent based on round robin algorithm 6. It makes sure to send refresh command to the memory every while to prevent the destruction of the stored data

7.2.3 Burst handler

It tell the arbiter to send a new burst if there is empty burst storage inside it by activating the start_new_burst_flag It stores the incoming requests in the right burst storage by deciding if the requests belongs to the current burst or it's a new burst to store It sends the bursts states to the timing controller then it takes the commands from the timing controller and convert it to DDR5 command based on the burst data stored inside then send it to memory It receives the data that came from the memory after sending the read command and stores the needed data in a special storage element to send it back to the returner afterwards It mappes the requested data to it's indices then ii sends them together to the returner to order it and send it back to the user or just the requests indices for the write requests

Chapter 8

DDR5 Controller Specifications

8.1 Address Mapping Scheme

The mapping scheme: After a lot of searching we found a lot of mapping schemes but further search revealed a paper named (permutation based page interleaving) that compares the permutation mapping scheme to several alternatives. It made it clear how it will be significantly better to use the permutation mapping scheme.

The permutation mapping scheme consists of two steps:

1. the first step is to order the bits to be in the most efficient way as if it is direct mapped.
2. then we use the xor operation between the least significant bits in the tag bits with the bank bits.

We have 30 address bits so we found out that is the optimum order is.

row	bank	column	Bank	group Column (least significant bits)
16 bit	2 bit	6 bit	2 bit	4 bit

Table 8.1: Address Format

Normally the read requests comes in groups and there is between them so to make sure we use the the memory in best way the order was like that:

1. The last 4 bits in the column bits are the least significant to make sure the burst request comes together to be issued in one command instead of 16.
2. The bank group bits follows to use the small timing between the bank groups between the burst and the next.
3. The rest of the column bits follows to make the next bursts in the same active row and avoid to activate any new row.
4. The bank bits follows because if we want to activate a new row any way then we will activate it in a new bank then we will avoid precharing the already activated row.
5. Lastly the row bits follows to make sure we use every storage bit in the memory.

Then we calculated the cache line tag bits no. for our last level cache and found out that it's 10 bits then we applied the permutation.

According to the paper mentioned above [4], the addresses that is mostly request together is located in different bank to use the memory system parallelism as possible.

The row miss rate is less than the half of the normal address mapping in the worst case and the performance get better as the no. banks increase until the miss rate reaches less than 10 percent.

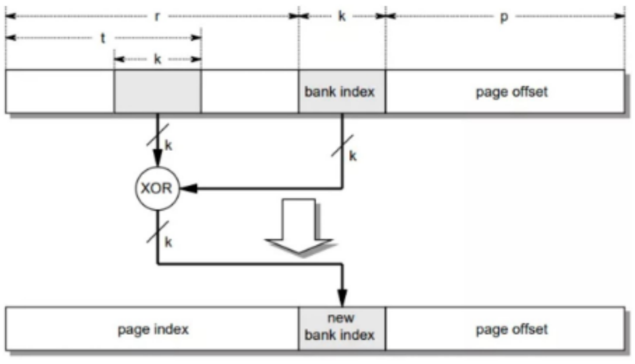


Figure 8.1: The permutation-based page interleaving scheme

8.2 Memory requests scheduling

DRAM is the most commonly used technology for building memory systems. However, it has been a main performance bottleneck for modern computer systems. Hence, many request scheduling algorithms are designed in order to reduce latency and exploit maximum row buffer locality. Exploiting row buffer locality in DRAM is a main key characteristic while designing proper scheduling algorithm for application needs. DRAM architecture is segmented into multiple banks to support concurrent accesses. Each DRAM bank consists of rows and columns of DRAM cells. Each bank is accessible with a row buffer, accessing it is faster than accessing different row in the same bank [5].

8.2.1 Scheduling Algorithms Trade-Offs

Deciding suitable request scheduling algorithm is a critical process. Each algorithm has its trade-offs, so we firstly analyze our application needs and environment parameters. Firstly, our environment applies (RoCE) protocol that is used in high capacity servers clusters, therefore, power is not a critical need for our application so we decided to use an open-row policy for draining bursts from bank schedulers. Secondly, we have level 3 associative cache in host node, hence, requests is received in terms of cache line size, so we seek for exploiting maximum row hits in bursts. Third, (DDR5) chip used in the system has an expensive time cost on switching the bus from read to write state, hence, minimum switching from read-typed to write-typed bursts and the opposite is needed.

During our studiying phase, we got three common scheduling algorithms and decieded which to choose for our application.

1. (FCFS) request scheduling
2. (RLDP) request scheduling [5]
3. Thread-Fair Request Reordering [6]

We found the following trade-offs mentioned below:

algorithm	switching between read and write bursts	hits
(FCFS)	High	medium
(RLDP)	medium/High	high
Thread Fair	medium/low	high

Table 8.2: Request Scheduling Trad-offs

(FCFS) algorithm has high rate of switching between reading and writing typed requests. It just keeps the same sequence of requests as they come in order. This may lead to bad throughput due to time consumtion of switching the bus. Regarding row hits, it may achieve some of it but it depends on the state of the incoming requests.

(RLDP) [5] has lower rate of switching than (FCFS), but it seeks for row hits as a main target so it may switch to write requests if there are available hits without keeping the switching rate as minimum.

Thread Fair Request Reordering [6] has lowest switching rate of them all im most cases, it always keeps looking for read typed requests as a main target before switching to write typed requests. Switching to write is done in short number of cases, only if they reach the (HWM) or there is no read typed requests available in the controller. Row hits are exploited but not as much as RLDP [5] do.

Our proposed controller uses thread fair [6] scheduling concept. We think this algorithm will fit our project requirements in both row hits and switching rate. In the next sections, we will discuss the block diagram of the bank scheduler itself and (FSM) of thread fair algorithm.

8.2.2 Block Diagram

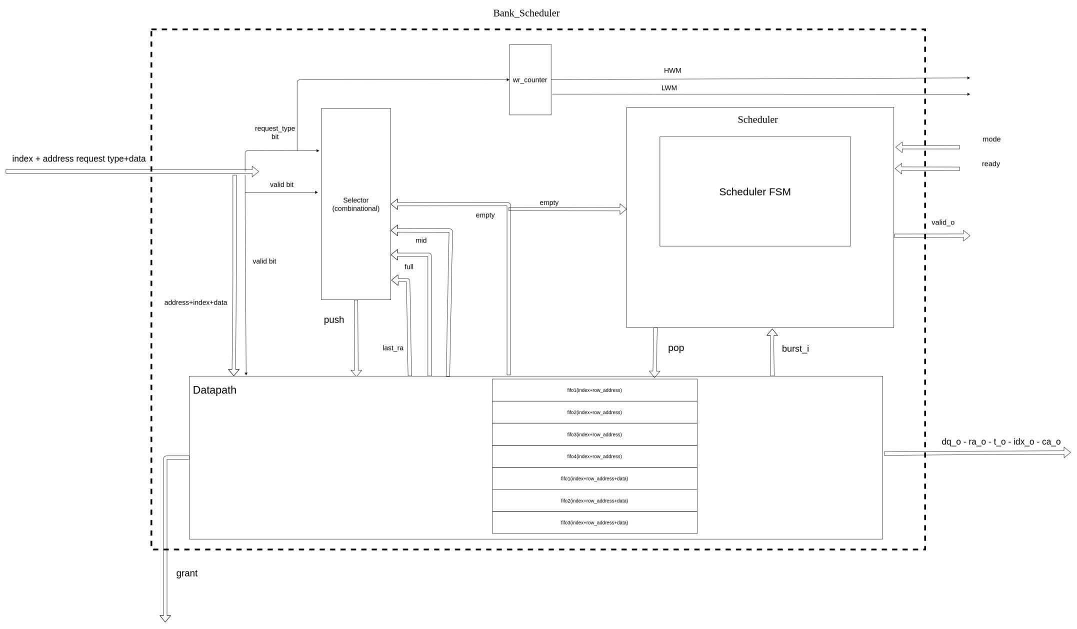


Figure 8.2: Bank Scheduler block diagram

Inputs:

1. Input Request
   - New requests come from transaction controller with address, data, index, type and valid bit.
2. Mode

- Signal from controller mode that determines current draining mode according to (HWM) and (LWM) Signals.

3. Ready

   - Comes from arbiter to complete hanshaking sequence.

Intermediate signals in block diagram:

Datapath provides full, empty and mid signals for both selector and scheduler parts. In addition, pop and push control signals are computed from scheuler and selector respectively into the datapath module.

Outputs:

1. valid

   - A signal from scheduler that guaranteeds stored requests are available. It is an important signal to start handshaking protocol with arbiter.

2. grant

   - Neccessary for indicating a successful reading from transaction controller.

3. output request

   - Next output request to be transmitted to arbiter from bank scheduler datapath

8.2.3 Selector

This block has the resposibility for filling in storage (FIFOs) with new coming requests according to a heuristic criteria that we chose for best allocation. As shown in the following figure, Selector part has some sort of greedy approach for filling in the new requests. In our project, most of new requests are in terms of cache line size, so it will be most likely has row hits continously. Selector tries to insert all matching row hits in a single (FIFO) for exploiting the advantage of our open-row policy in our (DDR5) chip. Using this criteria, all matching hits will be discoverd by the scheduler on the other side and maximum burst hits is achieved.

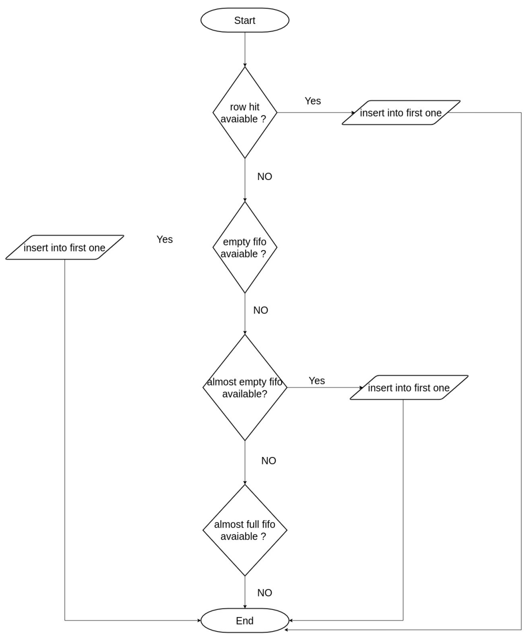


Figure 8.3: Selector Flowchart

8.2.4 Scheduler

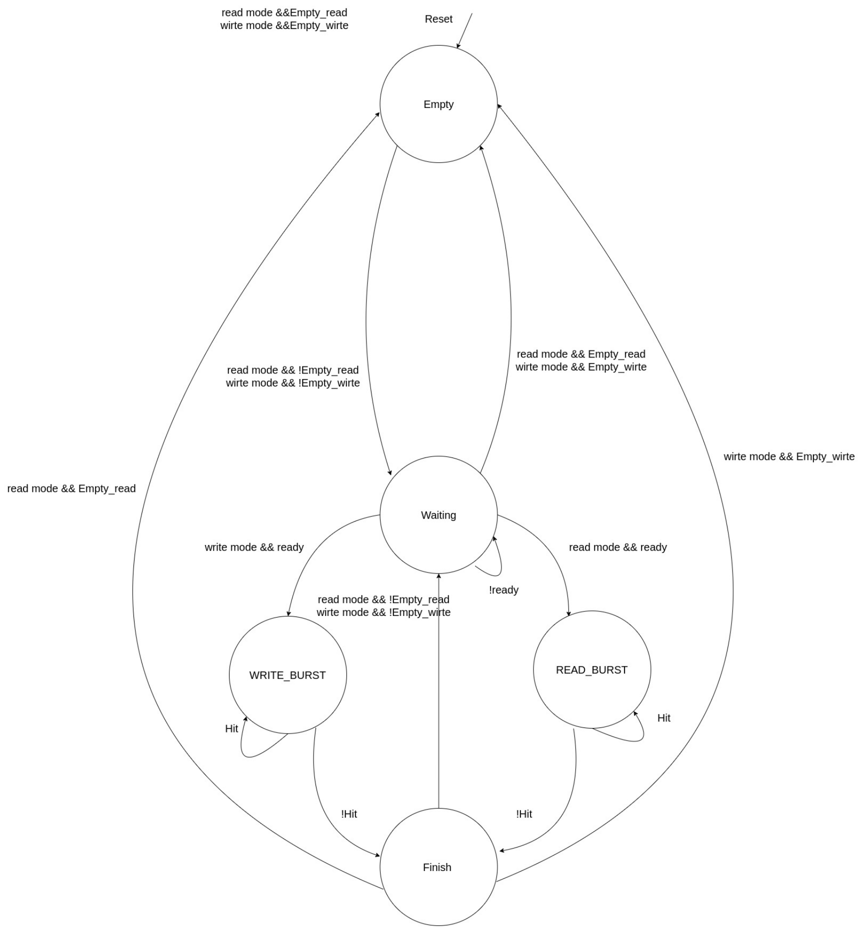


Figure 8.4: Scheduler FSM

As shown above, this (FSM) raises valid output to start the two phase handshaking with arbiter in “Wainting” state. The sequence of (FSM) is as follows:

1. After reset or empty (FIFO)s according to current controller mode, (FSM) starts with “Empty” state.
2. In case of new requests have arrived, a new transistion to “Waiting” state is done with raising valid signal as an aggreement for any new incoming ready signals from arbiter.
3. After access is granted from arbiter, scheduler starts draining the new burst according to current controller mode.
4. Scheduler only drains one burst with maximum number of sixteen requests per access and returns to “Waiting” state to wait for new grant access in the next cycles.

8.2.5 Datapath

We implemented this datapath for the sake of simplicity of design. All control signals from both selector and scheduler are inputs to datapth. In this way, modifications for both selector and scheduler are much easier without affecting the datapath through the top module.

8.3 Controller Mode

As we mentioned above, scheduling in our proposed controller has two modes. Mainly, controller starts in “READ” mode till (HWM) signal is set and the controller is switched directly to

"WRITE" mode. Since our goal is to minimize the switching time between write and read state in data bus with (DDR5) chip, controller mode is switching to write only if total write requests in the whole controller have reached a pre-define ratio specified as a paramter in "cntr_mode" module. Once our proposed controller has (LWM) signal with no row hits available, a switching to "READ" mode is done and draining read requests from schedulers starts.

8.4 Bank Arbitration

In memory controllers, arbitration is required in order to increase memory chips throughput by making best use of concurrent processes available and supported by our (DDR5) memory chip. Arbitration has many approches and designs. We will disscuss in the following sections how our proposed arbiter is designed, algorithm used to fit our application, in addition, flexibility and maintainability of it.

8.4.1 Concept Behind Our Arbiter Design

Before we discuss in such a deep way into design, we should show some important timing constraints in the new (DDR5) technology. There are three timing constrains we should take into consideration in the new (DDR5) technology as the following figure:

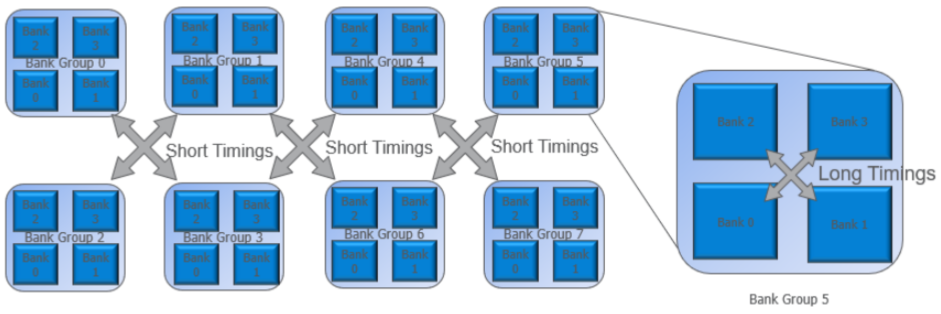


Figure 8.5: DDR5 Bank Access timings

1. Access time between two different bank groups
    - Arbiter must take in consideration this constrain, whatever the last accessed group is, arbiter should firstly select new different group to drain from.
    - Switching bursts between different bank groups has the shortest access timing.
2. Access time between two different banks in the same bank group
    - In case of only single bank group is ready, arbiter should choose new bank that is different from last accessed one in the same group.
3. Access time between different rows in same bank
    - The last decision available for the arbiter is to give access to same last access bank, this causes a row conflict and degrade the performance.
    - We can avoid this case by using effective scheme applier that can balance the quantity of traffic distributed in all banks.

Our proposed controller arbitration criteria aims to exploit maximum concurrent processess supported by (DDR5) bank groups technology, thus, our arbiter is aiming to grant access to controller data path to new bank groups if there are ready requests in it, Hence, exploiting maximum available concurrent accessing into a (DDR5) chip. Choosing a suitable arbitration sequence is so critical in order to increase the performance.

8.4.2 Block Diagram

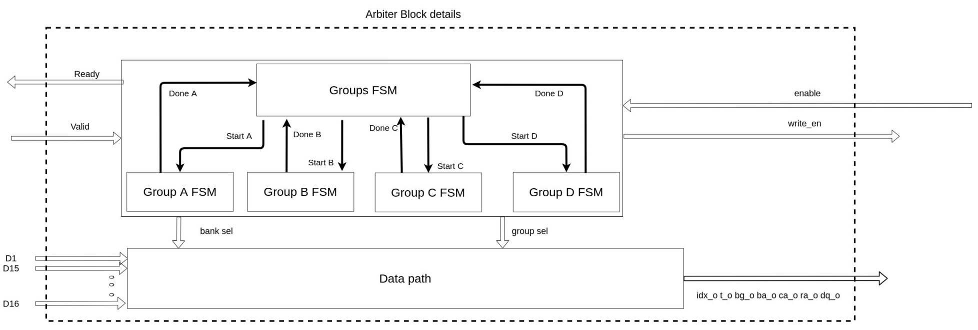


Figure 8.6: Arbiter block diagram

As shown above, we have hierarchical arbitration design that applies classic round robin algorithm [2] in both overall bank groups level and internal group level. The upper (FSM) has to round over all groups in order to exploit maximum parallelism accesses between groups. On the other hand, all lower (FSM) has to round over all banks in each group individually. Arbiter has three inputs:

1. Enable signal that enables arbiter is driven by backend part of our controller to receive new bursts into memory interface.
2. Valid signal that comes from bank schedulers to start handshaking with arbiter and synchronize the data transmission.
3. Data that comes from all bank schedulers to be forwarded into backend part by the arbiter.

Intermediate signals in arbiter:

1. Start signals that upper (FSM) sends to lower (FSM) to enable each group (FSM) individually in a round robin fashion
2. Done signals that are set to 1 when a new burst is a single group is transmitted successfully to backend part

3. Select signals from (FSM) hierarchy to forward the target burst from all banks to backend part

Output signals from arbiter:

1. Write enable signal to validate the current output data from the datapath
2. Ready signal output to bank schedulers in order to complete the handshaking communication between both of them
3. Data output from datapath to be transmitted into memory interface

8.4.3 FSM Design

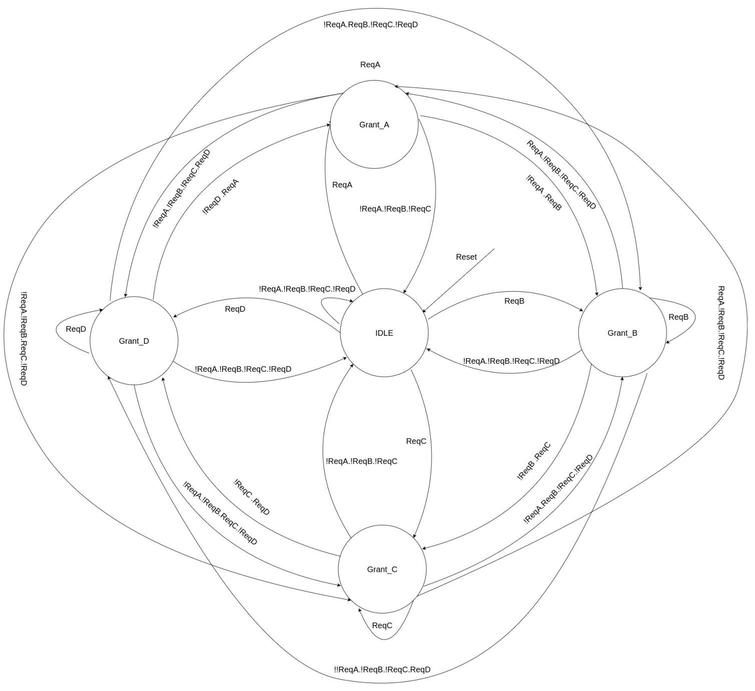


Figure 8.7: FSM diagram

This (FSM) design produces a pure round robin algorithm [2]. It is implemented in both upper (FSM) for bank groups and lower (FSM) for each internal bank group. In upper (FSM), the arbiter checks continously for new groups according to given valid signals and decide which group to go next. In lower (FSM), each one waits for its corresponding start signal to continue deciding which bank to go next [2]. Lower (FSM) triggers the output ready signals to the target bank.

8.4.4 Pros of Arbiter Design

This architecture has high flexibilty to edit and add features such as timeout, making round robin more adaptive to current traffic in all banks and other improvements. It also has high level of maintainability due to its simple hierarchy that provides easy interactions with other modules.

Scalability of this design is also a key factor of it, it can be expanded to support more bank groups according to memory chip of the system.

Chapter 9

Emulation Results [1, 2] [3]

edgdf		gfgf			fgfgfg
dfgd		dfgf			fgdggfgdg
fgf		gfgf			gfgf
		gf	fg		gf
fgf		gfg			

Chapter 10

Conclusions and Future Work

Bibliography

[1] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk.* Morgan Kaufmann, 2010.

[2] D. J. Smith, *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog.* Doone publications, 1998.

[3] W. Mauerer, *Professional Linux kernel architecture.* John Wiley & Sons, 2010.

[4] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, 2000, pp. 32–41.

[5] Y.-S. Moon, Y. Kwon, H.-S. Kim, D.-g. Kim, H. H. Lee, and K. Park, "The compact memory scheduling maximizing row buffer locality," in *3rd JILP Workshop on Computer Architecture Competitions: Memory Scheduling Championship, MSC*, 2012.

[6] K. Fang, N. Iliev, E. Noohi, S. Zhang, and Z. Zhu, "Thread-fair memory request reordering," *resource*, vol. 5, no. 8, p. 2.