DESIGN AND IMPLEMENTATION OF GENERIC MULTI-CHANNEL STORAGE CONTROLLER

A PROJECT REPORT

submitted by

MADUGULA SRINIVAS SANTOSH KUMAR

in partial fulfillment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY (DUAL DEGREE)
in
ELECTRICAL ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
MAY 2015

CERTIFICATE

This is to certify that the project entitled "**Design and implementation of generic multi-channel storage controller**", submitted by **M Srinivas Santosh Kumar**, in partial fulfillment of the requirements for the award of the degree of MASTER OF TECHNOLOGY (Dual Degree) in Electrical Engineering, at Indian Institute of Technology, Madras, is a record of bonafide work carried out by him during the academic year **2014-2015**.

Prof. V. Kamakoti
Designation and Guide
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai 600 036

Date: 11th May, 2015

ACKNOWLEDGEMENT

I owe my sincere gratitude to all those people who are responsible for the successful completion of this project and because of whom my graduate experience has been one that I would cherish forever. My deepest gratitude to my adviser Prof. V. Kamakoti whose energy, passion and support has been a tremendous source of inspiration throughout my project. I would like to extend my sincere thanks to my co-adviser G. S. Madhusudhan for guiding me through ups and downs of the project. I am fortunate to have advisers who gave me freedom to explore on my own and at the same time helped me to recover when I faltered. I would like to express my special thanks to Maximilian Singh, Sireesh and other project-mates at RISE lab for their valuable inputs and discussions. I am also very grateful for my friends for their technical inputs and making my life at IITM an exhilarating experience. I would like to thank my family for their valuable support and encouragement which made my life wonderful and I dedicate this project to my family and friends.

M S Santosh Kumar

ABSTRACT

With the increasing IO bandwidth requirements of the current day processing environment, the storage enterprise is rapidly moving towards flash storage technologies with NAND flash being the prominent among them. While NAND flash offer significant advantages over the traditional magnetic storage disks in terms of low access time, high throughput, reliability and IO parallelism, they also add additional complexities due to write after erase, limited number of block erase operations and bad block management. In order to efficiently handle these issues, a software stack called Flash Translation Layer (FTL) is traditionally used. Apart from handling these traditional issues, with the increasing shift towards multi-channel SSDs consisting of multiple channels, multiple-chips per channel, multiple dies per chip and multiple planes per die, FTL algorithms which take advantage of this underlying topology by exploiting the inherent parallelism plays a major role in the overall performance of the storage device. This resulted in various FTL algorithms with increased complexity, potential advantages and disadvantages of host-side FTL or device-side FTL implementations, page-based translation or hybrid translations etc. to be examined.

In this context, there is a need for a generic platform for research evaluations while maintaining commercial grade features for interoperability and extensibility and this project aims at providing such a solution. A generic multi-channel storage controller with parametric design variables and support for plug-and-play host-side or device side FTL implementations has been designed, implemented and evaluated. For increased maintainability and abstraction, it is implemented in versatile hardware description language *Bluespec System Verilog.* The implemented design is simulated and integrated with high performance, low latency bus interface, PCI Express, and emulated functionally using FPGA as a basis for the proof of concept.

Keywords*: Multi-channel storage controller, NVM Express storage, NAND flash memory, Solid state drive, generic storage controller.*

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Over the years, thanks to Moore's law and wide research community, processing power has almost increased 100 times over a decade, however, the storage IOPS have almost remained flat. As an old saying goes "A chain is no stronger than its weakest link", the same can be said about the performance of database systems. Magnetic HDDs have been the most common storage choice for database applications for over a decade. But given that the current day scenario of increasing cloud based applications, and thirst to increase cloud performance, looking to extract the performance from CPU clock cycles is no longer a viable solution, not when the current day CPUs can run at billions of cycles a second and rotating disks require milliseconds to complete an I/O operation. The storage is so far only measured in terms of capacity and performance through CPU clock cycles however as the "*storage gap"* continuous to increase, the storage speed exerts a tremendous impact on the performance. With the cloud applications sprouting all over the world, increasing trend towards cloud virtualization, all necessitate access to larger amounts of data in shorter amount of times, requires the problem of "*storage gap"* to be solved.

In this context, on one hand extensive research is being carried out in terms of developing techniques to avoid expensive random I/O, they could probably mask the poor I/O performance a bit but doesn't provide a permanent solution to the problem. On the other hand, huge amounts of research is focused on developing new storage technologies. Some of them include Phase Change Memories, Memristor technologies, NOR flash, NAND flash etc. Of these, NAND flash technology storage shows tremendous potential in meeting the current day requirements.

Apart from problem with I/O access speeds, HDDs also face challenges in terms of excessive power consumption. This is major problem in data centers where the power

consumption costs can be very high. In addition, the mechanical components of HDDs make them vulnerable to external shocks and increase maintenance costs.

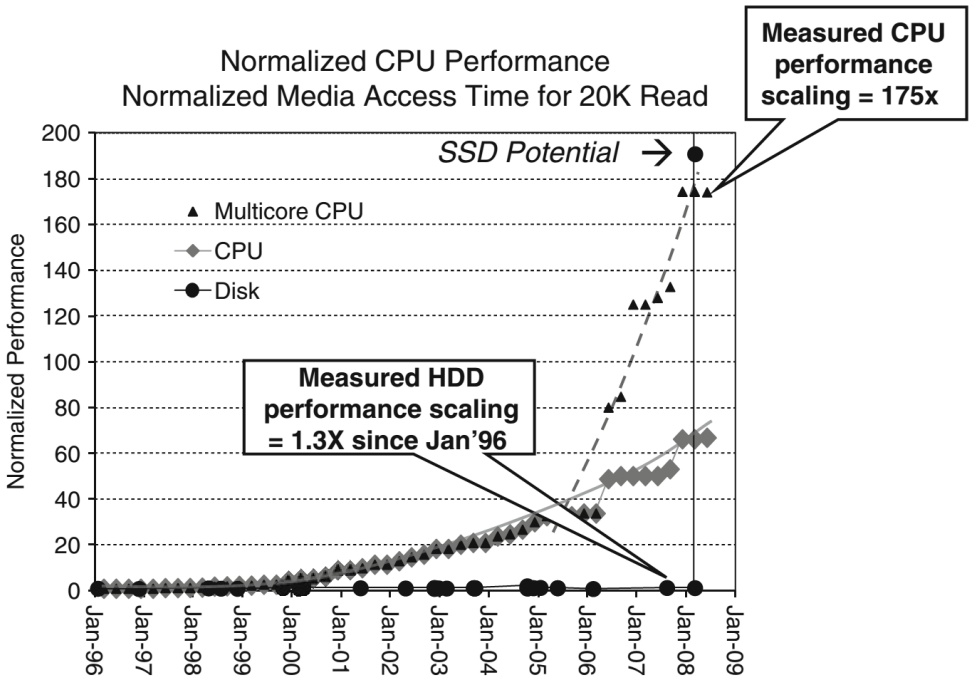

Figure 1.1: CPU vs HDD performance increase (Source: Intel measurements)

Fortunately, NAND flash technology, unlike HDDs, exhibit almost zero seek times, relatively lesser power consumption and are extremely resilient to external shocks. Figure 1.2 shows a comparison of SSDs with NAND flash technology and HDDs. The cost/byte of NAND flash is also coming down significantly with advances in fabrication technologies.

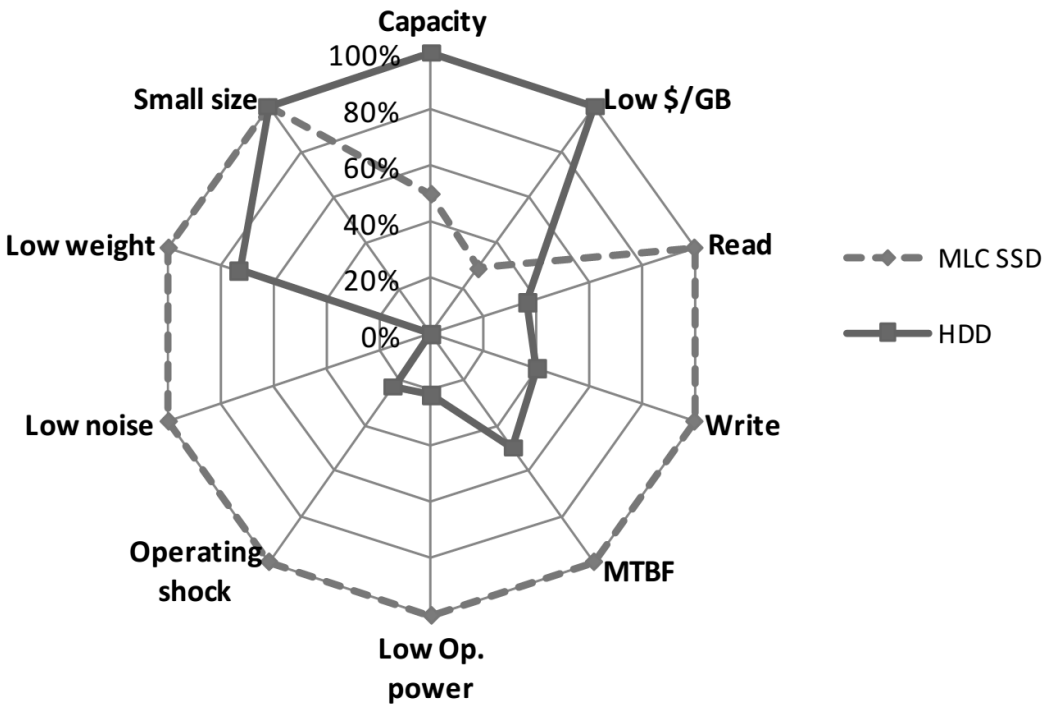

Figure 1.2: SSDs vs HDDs characteristics (Source: Forward Insights)

With so much market potential, could be well put in the famous words "there ain't no such thing as free lunch", NAND flash comes along with its own set of restrictions. In particular, flash chips have two-tier hierarchical structure consisting of pages and blocks. Each block consists of a certain number of pages and all writes and reads are exhibited at the granularity of pages. However, unlike traditional storage media, these pages cannot be over-written, they needs to erased before a new set of data is to be *programmed* and this erase can happen only at the level of block. In addition, the erase latency is much higher when compared to the *program* latency. Also, there are only a limited number of erase cycles before failure of the block. All these new restrictions resulted in extensive research to exploit the full potential of NAND flash technology. Now, this places a huge significance on storage controller architecture and a need for an open-source research platform. In these lines has been launched the project *Lightstor*, aimed at building a complete *Solid state development kit* with support for state of art protocols, interconnects, parametrized design variables, modularity and extensibility.

When it comes to storage technology, the host interconnect technology and topology also plays a major role in determining the overall performance and hence *LightStor* storage controller has generic host interfaces. In this thesis, the storage controller is developed with PCI Express interconnect, however, *Lightstor* aims at combining the storage controller with *RapidIO* as well and *LightNVM*, which is an extended version of NVM Express specification supporting both host-side and device side *Flash translation*.

The designed controller fits naturally into both the interconnect technologies and is tested with PCI Express interconnect. It also supports both device and host-side *Flash translation* interfaces and hence combined with *LightNVM* provides a complete research platform for evaluating the advantages and disadvantages of various architectural techniques.

Thesis Overview

The rest of this thesis is organized into the following phases.

Background

- Familiarization of concepts in NAND flash technology.
- Familiarization of host interface protocols like NVM Express and PCI Express, traditional design principles and literature review.
- Introduction to high-level hardware-description language *Bluespec system Verilog* and reason for the choice.
- Familiarization with Linux device drivers and kernel modules.

Conception and scope

- Introduction to the concept of generic multi-channel storage controller
- Introduction to system-level architecture and definitions of blocks from the perspective of storage controller.
- Scope of this thesis

Architectural design

- Design and implementation of generic multi-channel NVM controller.
- Design and Implementation of PCI Express wrapper

Testing and evaluation

- Design of test environment and FPGA emulation techniques.
- Functional testing and evaluation results.
- Synthesis results and optimizations

Extensibility and Future work

- Design choices and ways of extending generic controller to other use cases.
- Summary and future work

CHAPTER 2

BACKGROUND

This chapter introduces the basic concepts that are necessary for understanding the following chapters. It is also essential for a storage controller architect to understand the underlying NAND flash technology and intrinsic behavior in order to better tune the architectural features for unleashing the complete potential of NAND flash.

Since NAND flash technology has been around for years now, there is much to learn than that can be said in this thesis. However, an abridged version of concepts have been presented at various levels starting from NAND flash cell technology, why's and what's of current day NAND flash architecture, their integration with storage architectures, host protocols, and specifications

2.1 NAND FLASH TECHNOLOGY

Flash as a memory technology has been around for more than two decades now. Flash has been invented by Dr. Fujio Masuoka of Toshiba corp. in 1984. Following shortly, Intel has commercialized NOR flash memory as storage medium for storing BIOS and firmware for various consumer products. Although flash technology started with NOR topology, the NAND topology has been found to have tremendous benefits, as we will see shortly. Following 2009, NAND flashes have found its way into various storage devices including memory cards, USB flash drives, solid-state drives and even as non-volatile caches. The major impact of NAND flash as a storage technology is in the replacement of magnetic HDDs with Solid State Drives (SSDs). While the concept of SSDs have been around for a while, only the capabilities and ever-lowering cost of NAND flash technology has made it finally viable. The following sections will look at flash as a cell and build towards NAND flash chips, including their advantages and restrictions followed by the storage controller design.

2.1.1 Flash cell

A flash cell is made of a floating gate transistor which resembles a conventional MOSFET except that it has two gates (see Figure 2.1). A primary gate (or) control gate which is electrically isolated by a surrounding oxide insulator creating a floating node, hence the name "*floating gate transistor"* and a secondary gate which is connected to the gate terminal is only capacitively coupled with the floating gate. The floating gate behaves as an excellent electron "trap", which can retain charge for years because of complete isolation and this electron trap constitutes a memory bit. The act of injecting and removing electrons from the floating gate are called *program* and *erase* respectively.

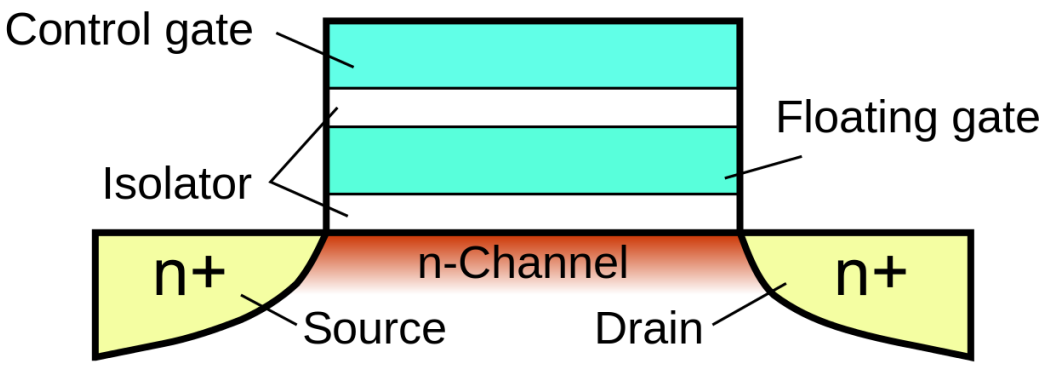

Figure 2.1: Floating gate flash memory cell [1]

The act of *programming* or *erasing* is carried out using *Fowler-Nordheim Tunneling*. When a strong electric field is applied across the oxide, due to quantum mechanical tunneling, the electrons tunnel through the *oxide tunneling barrier* and gets trapped on the floating gate. The presence and absence of the charge on the floating gate results in different threshold voltages of the memory cell. This can be sensed by applying voltage with value between these threshold voltages on the secondary gate and test for conduction of channel which determines the logical bit stored in the memory cell.

This charge trapping mechanism raises two important reliability issues, *charge retention* and *degradation.* The problem of *charge retention* arises due to multiple leakage paths like leakage through secondary gate, cell side wall oxide etc. The process of tunneling *degrades*

the oxide layer leading to certain number of erase cycles before it becomes usable. Typically, NAND flash cells have 10 years of charge retention and 1k to 100k program/erase cycles.

2.1.2 Flash organization

The flash cells are typically organized into memory arrays of two different topologies. NOR topology and NAND topology. NOR topology is shown in Figure 2.2 and NAND topology in Figure 2.3. From the application perspective, the major difference between these topologies is the *access granularity* and *access time*. NOR flash are byte/bit addressable and allows applications to run directly, similar to main memory, and are typically used for code storage as in BIOS or as NVRAM. NAND flash, on the other hand, doesn't allow fast random access at byte-level, are typically accessed in pages and access time for the first byte is quite higher when compared to NOR flash while the sequential read from the same page is faster.

Though the NOR flashes allow for fast random access, they have high erase times, larger area, higher cost per byte, lesser maximum erase cycles and high power consumption. While NAND flash on the other hand has much lesser area with higher integration density and hence lesser cost per bit and lesser power consumption. This makes NAND flash more suitable for secondary storage such as *Solid state disks* and the initial slow random access is not a strong drawback because typically data is accessed in large sequential blocks by processing systems. This is because of paging mechanisms in memory structures, exploiting locality of reference in main memory. From here on, we only refer to NAND flash topology.

The memory cells in NAND flash are placed in a matrix to optimize the silicon area. Logically, a NAND flash is organized into *pages* and *blocks*. A *block* is the smallest erasable unit. A block is divided into multiple *pages* and *page* is the smallest addressable unit for reading and writing. Each *page* is composed of *main area* and *spare area*. The *main area* is typically of size 4 or 8 KB which is used for data storage and *spare area* is of the order of hundreds of bytes per page and typically used for storing ECC information and system pointers.

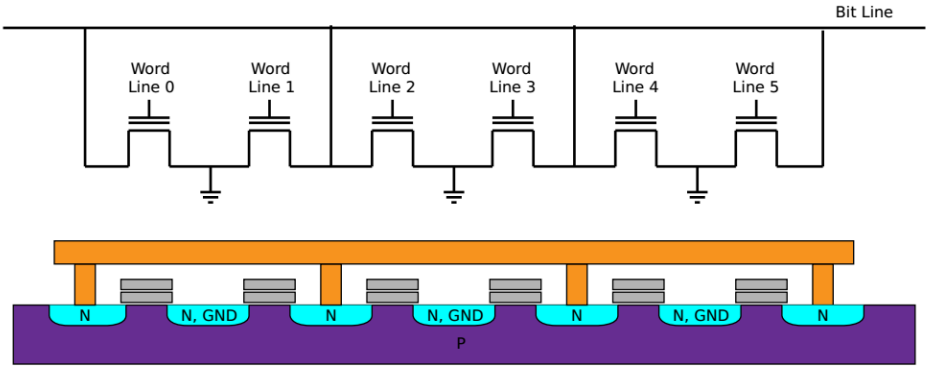

Figure 2.2: NOR Flash structure [2]

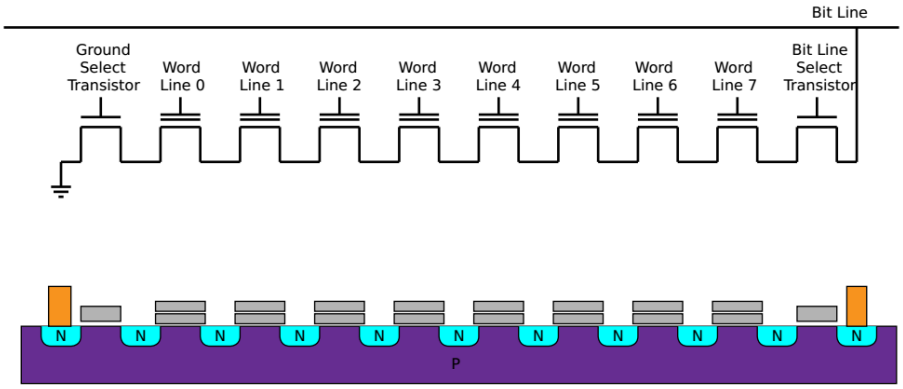

Figure 2.3: NAND flash structure [2]

Because of this particular organization, write-after-erase and limited number of program/erase cycles per block, a storage controller is needed to ensure correct operation and retrieval of data.

The NAND flash is also characterized by the number of bits they hold per cell. SLC (Single Level per Cell) flashes hold 1 bit of information per cell whereas MLC (Multiple Levels per Cell) flashes can hold more than one bit of information per cell. MLC increase the integration density and hence reduce cost/bit, however, they have higher access times, lesser erase cycles and higher power consumption.

2.1.3 Recent developments in storage technologies

One of the recent development is to replace the *Floating gate transistor* with *Charge trap flash*. This technology uses silicon nitride as a trap layer instead of a floating gate made of poly silicon to increase the integration density and number of write-erase cycles. It however has a more complex fabrication process and higher cost as of now. One interesting development in increased integration density is 3D flash allowing the flash cells to be fabricated across various silicon layers on a single chip instead of a single 2D silicon layer.

Also, current research is focused in exploiting other storage technologies like *magentoresistive RAM, phase-change memories and millipede memories. Phase-change memories* show tremendous potential in replacing flash in the near future, however it hasn't yet achieved much commercial success.

2.2 NAND FLASH STORAGE CONTROLLER

Now that we are familiar with the flash organization, in order to see the need for keenly architected storage controller and techniques, let us consider a naïve approach in which a flash chip is provided as is and the controller simply translates the host interface commands into flash reads and writes. In particular, let us consider a page write. Since we cannot overwrite any page, each write would require an erase and program of the corresponding page and since erase can happen only at the block level, all other pages in the block have to be read and written back following the erasure. Given that an erase block would contain 128 pages or

more, such an overhead is easily intolerable. Also, in case of power failure, the data that is read out pre-erase could be potentially lost. In addition, given limited erase cycles, such a strategy would lead to permanent failure of the blocks containing pages which are frequently written and in this strategy once few of the blocks fail, the NAND flash is rendered useless given the static address mapping between host addresses and physical addresses.

These problems with this naïve approach are very quickly attended to by the research community and techniques that are used to overcome this are presented below.

2.2.1 Dynamic address translation

There is one major problem with this naïve approach i.e., the complete erasure of a block even when we are writing only to a single page. This problem can easily be solved by replacing the static mapping from the host addresses to physical NAND flash addresses with a "*persistent"* dynamic mapping. In this approach, if a write needs to be executed, a clean page is located on the flash and the data is directly *programmed* into it and the persistent map is updated. Note that the map has to be "*persistent*" and again this leads to another challenge in terms of how you would maintain such a "*persistent"* map. Also note that this map has to be accessed on every write. So, accessing the map from flash is not an option, considering high access time. Traditionally, this map is maintained on SRAM/DRAM memory and the persistency is maintained by writing them back to NAND flash on shutdown and bringing them into memory on power on. This would require the memory cells to be equipped with a battery to handle situations of power failure. However, this approach with *page-level address translation* would require large amounts of DRAM cells increasing the silicon area, power consumption and cost. Hence several space-efficient implementations such as *Block Associative Sector Translation (BAST)* [3], *Fully Associative Sector Translation (FAST)* [4] have been proposed which does some of the translations at block level and some at page level. This hybrid approach though not as efficient as page level mapping can reduce the precious silicon space. Techniques like *Demand-based flash translation layer (DFTL)* [5] can be used to reduce the storage space while still using page-based mapping and also improve the NAND flash lifespan.

2.2.2 Garbage collection

Of course, the dynamic address translation doesn't prevent us from block erasure. We will need to erase the blocks at some point to continue writing new or modified pages. This process is called *Garbage collection*. Once a logical page is re-mapped, the older page is marked for erasure. The algorithms to execute garbage collection is very important in maintaining performance and endurance of NAND flash. Naïve approach, as we have seen, results in huge *garbage collection overhead*, since an entire block has to be erased for every page write, and there has been extensive research which takes garbage collection overhead also into account during translation to yield better performance. Some of the attempts can be found in *Fast and Endurant Garbage Collection (FeGC)*[6], *Garbage Collection Algorithm based on Area and Block (GCbAB)*[7] etc.

2.2.3 Wear levelling

Wear levelling is the policy of distributing the block erases evenly across the NAND flash in order to keep all the blocks of NAND flash alive as much as possible. Note that simple "*out of place*" updates considering the history of block erasures need not necessarily be an efficient wear levelling algorithm because such an algorithm would not handle *static data*, the data that is not frequently updated. The blocks with frequently updated data and blocks with static data co-exist in typical workload conditions resulting in certain blocks to be erased frequently than others. Hence the blocks with *static data* would need to be moved to ensure proper *wear levelling*. This would result in wear levelling also to do extra data movement and wear levelling overheads are important. Significant research has been focused on this as well, techniques like hot-cold swapping [8], evenness aware algorithm [9] etc. are proposed.

2.2.4 Bad block management

Bad blocks are those which have failed and not suitable for further data storage. This could happen during manufacturing of NAND flash itself or during NAND life time because of limited erase endurance. It is one of the reasons why *Error Correction Codes (ECC)* has to be implemented on the NAND flash controller to detect and correct errors and marks block as bad block if the error is uncorrectable. A persistent bad block map, typically a bit map, is also maintained inside the controller for this purpose.

All the techniques described above are collectively referred to as *Flash Translation Layer (FTL)*. Since this is inherently software, now the obvious question to ask is where to implement the FTL. Whether it needs to be done on the device-side i.e., inside the storage controller or on the host-side? Before we answer this question, we will need to look at another aspect of *FTL* corresponding to I/O parallelism in NAND flash.

2.3 I/O PARALLELISM AND MUTLI-CHANNEL CONTROLLER

With the ever increasing thirst for I/O speeds especially in this world of overflowing data, simple replacement of magnetic HDDs with flash wouldn't meet the demand, we need high bandwidth flash array architectures. Also, with the host I/O interfaces like PCI Express and RapidIO providing huge bandwidth, the bandwidth of a single flash chip is no match to the bandwidth offered by these interfaces. Operating a single flash chip connected to such high bandwidth interfaces would simply be a wastage of power and area especially in *SSDs*. Note that increasing the number of flash chips per package would increase both the integration density and read/write throughput. Hence it is natural to go for multi-chips operating in parallel. There are couple of ways to achieve this.

2.3.1 Die level parallelism and interleaving

The idea is to increase the number of dies per channel as shown in Figure 2.4. And then the operations on a single channel can be interleaved i.e., a second chip on the channel can be addressed when the first one is busy. For example, the write operations to a single channel to various dies within in the channel can be interleaved as shown in the Figure 2.5. The amount of parallelism in this approach is of course limited by the *program time* in case of write but way before that, this approach is limited by the channel parasitic loading and hence doesn't give complete write parallelism. Typically, there would be 2/4 dies per channel.

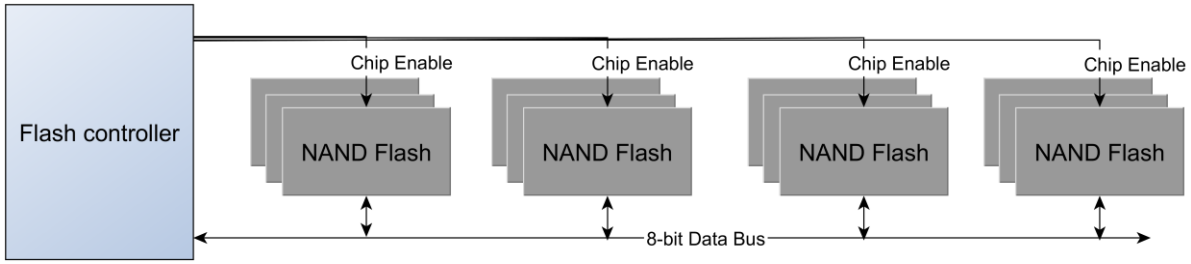

Figure 2.4: Multiple die connected to single flash channel

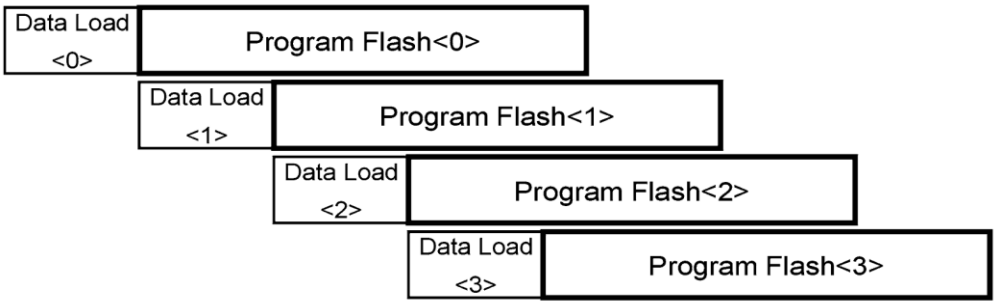

Figure 2.5: Interleaving within a flash channel

2.3.2 Channel level parallelism and multi-channel controller

Here the idea is to increase the number of channels as shown in Figure 2.6. This approach will shift all the complexities into the memory controller which now needs to handle multiple requests and data coming from multiple channels. This solution is however much more scalable and flexible than the previous one. Now what prevents us from increasing the number of channels indefinitely? Of course, the power consumption. Multiple channels operating in parallel draw huge amounts of current in parallel, increasing the power consumption.

In practice, both the solutions are used within a given chip i.e., multiple channels per controller and multiple dies per channel. In addition there are other kinds of parallelism like plane level parallelism within in a die. This however allows limited amount of parallelism because of several restrictions.

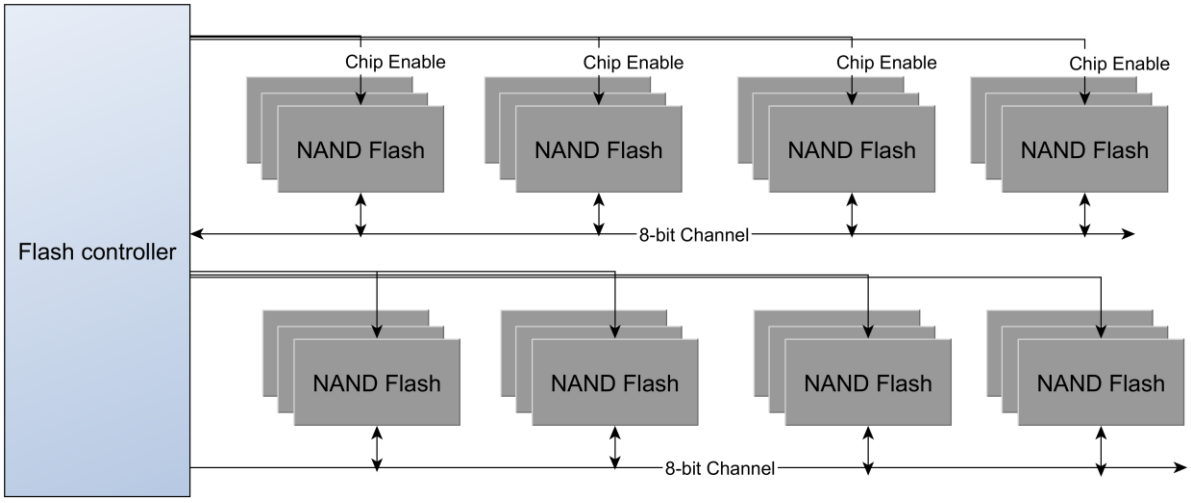

Figure 2.6: Multi-channel flash controller with multiple die per channel

Now that, we have seen the inherent I/O parallelism offered by the current day flash memory architectures, in order to fully utilize the offered bandwidth, the distribution of I/O access patterns of the underlying NAND flash plays a very important role in determining the performance. Hence apart from the functions described in the earlier section, it is now also the role of *Flash Translation Layer* to distribute the I/O requests across the channels and across the chips so that multiple chips operate as parallel as possible.

2.3 HOST-SIDE vs DEVICE-SIDE FTL

The *Flash Translation Layer* is now responsible for *dynamic address translation, garbage collection, wear-levelling, bad-block management* and *I/O parallelism*. Efficient handling of all of them plays a very important role in determining the overall performance of an SSD. Naturally, as *FTL* algorithms evolve, there will also be an increase in the complexity and hence the increased resources required for handling the FTL. In this context, we examine the potential advantages and disadvantages of handling FTL on host-side and on device-side.

- Availability of tremendous computing resources and buffer space at the host side increases the flexibility of the FTL to incorporate more efficient and complex algorithms. For eg. Page mapping FTL, which is more efficient than block mapping

or hybrid mapping FTL but requires larger buffer space for mapping tables, is easy to implement on host-side than on the device-side.

- Implementing FTL on host-side would consume precious CPU resources that would ideally be handling more IOPS.
- Direct access of host-side FTL to incoming I/O request information enables the host-side FTL to exploit inherent IO parallelism in SSDs more efficiently. See *Parallel Issue Queuing* [10] for an example on how host can schedule conflict-free I/O requests before reaching the controller.
- Dynamic channel contention aware data scheduling is not directly possible with host-side FTL without real time communication of device internal contention with host-side stack. Real time communication with host side is difficult to achieve due to high latency of host-device communication interface.
- Host-side FTL can provide operating system support for various storage interfaces going beyond the traditional block device interface such as key-value stores, atomic IO access etc. [11] providing storage efficient APIs especially to cloud application vendors.
- Device-side FTL has the advantage of hiding the flash internals from the host and providing simple read/write interfaces to the host where as host-side FTL requires modifications to the OS layer and raises OEM qualification issues.
- Host-side FTL has the advantage of keeping storage controller architecture easier and reducing the cost of the overall storage chip while it might not deliver a bootable drive because the system must be booted for flash-management software to execute and enable the storage device.

Despite the above advantages and disadvantages, there is still much research to be done on this front, while most of them are use-case specific. In this context, it is essential to have a generic multi-channel controller which supports both host and device side FTL. The current thesis combined with *LightNVM*, which we will discuss shortly, are attempts to build such a controller. Also research oriented towards hybrid solutions is essential and this thesis aims at providing a research platform for the same.

The remainder of this section looks at various host interface protocols needed for development of this generic multi-channel controller.

2.4 NVM EXPRESS SPECIFICATION

NVM Express (NVMe) [12] is a scalable host controller interface designed as a standard protocol for next generation non-volatile storage media. The *NVMe* originally intended for PCIe based SSDs, however is almost independent of the kind of interface and can be easily extended to interfaces other than PCIe. The *NVM Express* abstracts the device internals to the host through simple read/write commands. However, if we want standardized flash controllers which support host-side FTL, additional commands are to be added on top of the standard NVM Express specification. This exactly is done in *LightNVM* specification which supports NVMe devices and also provide extensions for host-side FTL implementations.

In this section we will discuss the general flow and command structure of NVM Express compatible device and in the following section we will discuss the additional command support added in *LightNVM* to enable host-side FTL implementations.

2.4.1 General command flow in an NVMe device

Figure 2.7 shows the general command flow in NVMe device. NVM Express uses main memory as a common communication point between host and the controller for most of its operations. All the commands and data reside in the main memory and a new command or data arrival into the main memory is merely notified to the controller through memory mapped configuration registers.

All the commands follow a paired submission and completion queue mechanism. Command queues are simply circular buffers with a fixed slot and reside in the main memory. Commands are placed by the host into a submission queue and completions are placed by the controller in the completion queue. Multiple submission queues may utilize the same completion queue. A maximum of 64 I/O queues is supported by the specification however the controller can support only a few which is advertised by the controller capabilities. Among these queues there are a pair of special queues called *Admin submission queue* and *Admin completion queue.* They exist for the purpose of controller management and control (for eg: creation and deletion of I/O queues, aborting commands etc.) and there is a special command

set called *Admin command set* which may only be submitted to *Admin submission queue.* Similarly there is an *I/O command set* which can be submitted to any of the *I/O submission queues.*

Whenever the host wants to send an *Admin command* or an *I/O command* to the controller, it places the command in the respective submission queue (SQ) and updates the submission queue tail doorbell of the corresponding submission queue, which resides in the one of memory mapped registers of the controller. In this way, the controller gets notified about the existence of a new command in the corresponding queue and then the controller can fetch the SQ entry in order from the SQ but there is no guaranteed order in the execution of the commands, the commands can be executed out of order.

Every SQ is associated with a completion queue (CQ), which is specified during the creation of the SQ using *create submission queue* command. Multiple SQ can be associated with a single CQ. Whenever the controller completes the execution of a command, the controller writes a *command completion* into the CQ and lets the host know through an *interrupt* or *phase bit*.

A *phase bit* is part of the CQ entry which is flipped whenever the controller has wrapped around the top of the completion queue (remember that completion queue is a circular buffer). Initially the host sets the phase tag values for all the completion queues to be 0 and when the controller goes through the first round, it updates the phase tag with 1 and once it goes through the next round, it updates it with 0 and so on. The host can know about the new completion entry by polling the *phase bit* or through an *interrupt*. Once the host processes the completion queue entry, it updates the *completion queue head doorbell* which resides in the controller memory mapped registers to let the controller know that the entry can be reused.

2.4.2 Configuration registers

The configuration registers are mainly used for two purposes: setting the state of the controller and notifying the controller about new commands in the main memory. These registers can be directly accessed by the host, memory mapped to the host address space and are un-cacheable. Table 2.1 outlines the controller registers and their functions.

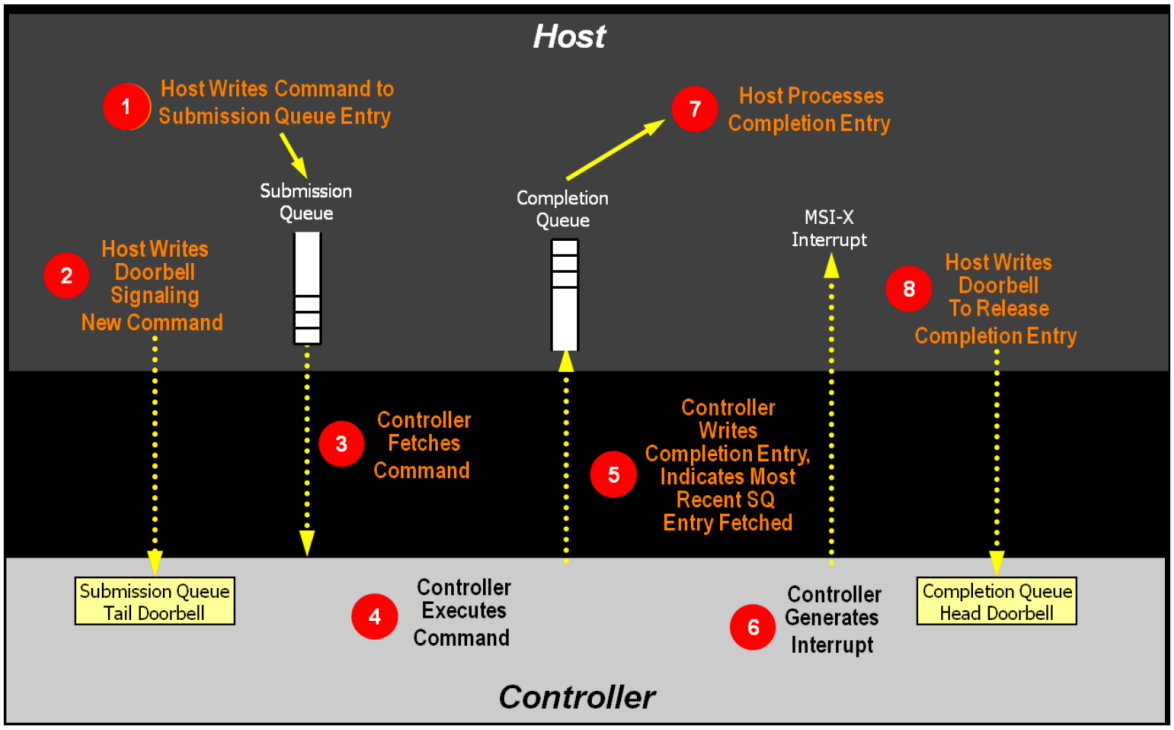

Figure 2.7: General command flow in an NVMe device [12].

Table 2.1 Configuration registers of an NVMe device

Controller register	**Function**
Controller capabilities	Indicates basic capabilities of the controller to the host, like max and min page size, supported command sets, arbitration mechanism etc.
Version	Indicates major and minor version of NVMe spec that the controller supports
Interrupt Mask set	Used to mask interrupts for specific CQs
Interrupt Mask clear	Used to clear previous set mask

Controller configuration	Used to modify controller settings based on *controller capabilities* register
Controller status	Holds the current status of the controller
NVM subsystem reset	If supported, it is used to reset the controller
Admin Submission queue base address	The base address of admin SQ in main memory.
Admin completion queue base address	The base address of the admin CQ in main memory
Submission queue doorbell registers	To inform the controller about new commands.
Completion queue doorbell registers	To inform the controller that the completion in the main memory are processed by the host.

2.4.3 Controller Admin Command set

The controller admin command set is used for setting the state of the controller, get controller features and controller initialization. Table 2.2 briefs the admin commands defined in the NVM Express specification.

Table 2.2: Admin Command set for an NVMe device

Admin command	**Description**
Abort	Used to abort specific commands previously submitted to admin or I/O SQ. It is the best effort by the host and doesn't guarantee an abort by the controller.

Asynchronous Event Request	Control asynchronous status report of the controller.
Create I/O completion queue	Creates a new completion queue
Create I/O submission queue	Creates a new submission queue.
Delete I/O completion queue	Deletes an existing completion queue.
Delete I/O submission queue	Deletes and existing submission queue.
Firmware Activate	Used to verify that a valid firmware image is downloaded and to commit that revision to a specific firmware slot.
Firmware Image download	Used to download all or portion of firmware for future update of controller.
Get features	Retrieves the attributes of the features specified
Get Log page	Get extended status info from the controller
Identify	Get information regarding the controller, namespaces etc.
Set features	Set controller features
I/O command set specific	Additional defined command sets
Vendor specific	Additional commands defined by the vendor.

2.4.4 NVM I/O command set

The I/O command set is targeted towards the underlying non-volatile memory. Table 2.3 briefs the I/O commands specified in NVM Express specification.

Table 2.3: I/O commands set for an NVMe device

NVM I/O command	**Description**
Flush	Flush the contents of cache to underlying non-volatile memory
Write	Write data from main memory to non-volatile memory.
Read	Read data from non-volatile memory to main memory.
Write Uncorrectable	Used to mark a logical block as invalid. Any further reads from this block returns an error. However, to clear invalid logical block status, a write has to be performed on the logical block.
Compare	Compare logical blocks of memory without moving the data to host.
Write zeros	Write zeros to main memory block.
Dataset Management	Assign attributes like access latency etc. to a block range.
Reservation related commands	It is used in multi-controller environment to acquire and release namespace registrations. A *namespace* is set of non-volatile memory which can be formatted to logical blocks.
Vendor specific	Additional commands which can be extended by the vendor.

Of all the commands specified above, *Read, write* and *flush* commands are mandatory and the rest are optional.

2.4.5 Address exchange – Physical region page and scatter gather list

A single I/O command can specify multiple pages of data transfer. The size and structure of the I/O command is fixed, however multiple page transfer requires multiple main memory addresses and multiple NAND flash addresses to be specified. In NVM Express, the NAND flash logical addresses in a given I/O command are contiguous and hence only require a base address and length. Therefore, these fields are fixed and included in the command structure itself. However, the main memory addresses need not be contiguous and hence requires *address lists*. These addresses are specified outside the command structure and the command only consists of a pointer to these structures. There are two ways of specifying these address lists in NVMe.

2.4.5.1 Physical region page (PRP) entry and list

A Physical region page (PRP) entry is a pointer to the main memory page. These are fixed 64-bit entries consisting of *page base address* and *offset* as shown in Figure 2.8 .The *offset* is zero for all PRP entries except for the first PRP entry in the command or PRP list.

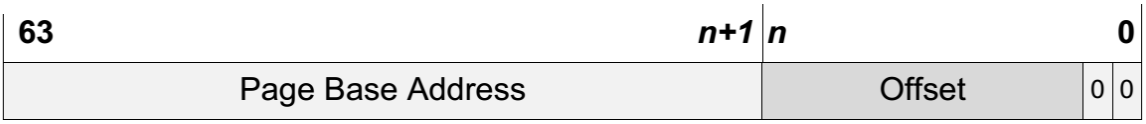

Figure 2.8: Layout of PRP entry [12]

The value of *'n'* in the Figure 2.8 depends on the size of the memory page

A PRP list is a set of PRP entries in a single page of contiguous memory. The list identifier itself is a PRP entry pointing to the PRP list residing in the main memory. There are certain address fields available within the command as we will see in the next section. PRP lists is used to specify the addresses if the data transfer could not be described in the command itself.

2.4.5.2 Scatter Gather List (SGL)

A SGL is a data structure in main memory used to describe a data buffer without any alignment requirement. An SGL consists of segments and each segment is a Qword aligned data structure in contiguous region of main memory describing a complete data buffer or additional segments. Basically, it forms a linked list kind of description of main memory buffers. For exact details of SGL data structure refer to [12]. These descriptors allow a more flexible description of data location however it adds complexity and increased overhead to the underlying controller because of its complex structure.

2.4.6 General Command structure

The commands in NVM Express are a fixed 16 DWord data structures. The general command structure is described below.

DWord 0

Command Dword0 consists of four fields.

1. **Command Identifier:** A unique identifier for the command when combined with SQ identifier.
2. **PRP or SGL transfer:** This field specifies whether PRPs or SGLs are used for any data transfer associated with the command. All the *admin commands* will use PRPs.
3. **Fused operation:** In fused operation, a complex command is created by *fusing* two simpler commands. The field specifies if the commands is a fused command or not.
4. **Opcode:** This field specifies the opcode of the commands to be executed.

DWord 1 (Namespace Identifier)

This Dword specifies the namespace ID that this command corresponds to. If no namespace ID is used for this command, this field is cleared to 0.

DWord 2 – Dword 3: Reserved

DWord 4 – Dword 5 (Metadata pointer)

This field contains the address of a contiguous physical buffer of metadata.

DWord 6 – Dword 7 (PRP Entry 1)

Specifies the first PRP used for the command.

DWord 8 – Dword 9 (PRP Entry 2)

This field specifies second PRP, this is either a PRP address or a pointer to a PRP list used for the command. Since the command has two PRP entries, if the data transfer corresponds to more than two PRP entries, than this field specifies a pointer to PRP list, else it specifies a PRP entry.

DWord 10 – Dword 15 (PRP Entry 2)

Command specific data. Refer to [12] for detailed description of this field.

2.5 LIGHTNVM SPECIFICATION

LightNVM specification extends the NVM Express specifications to support host-side FTL implementations. It allows the controller to off-load all or a subset of the features of *Flash Translation Layer (FTL)* like logical to physical address mapping, persistency of page tables etc. and also other functionalities like ECC for flash. To support this functionality, the command set is extended. Some of the extended commands are provided below.

2.5.1 Extended admin commands

Table 2.4 gives the description of the extended admin commands.

Table 2.4: LightNVM extended admin command set

Extended admin command	Description
Set responsibility command	It allows the host and device to negotiate its responsibilities like hardware ECC vs ECC by host, device-side FTL vs host-side FTL.
Get Logical to Physical Translation table	The controller returns a range of logical to physical translation table to the host.
Get Bad blocks table	The controller returns a bit map of bad blocks.
Set bad block table	The host sets the bad block table entries.

In addition to the new commands described in Table 2.4, the *Identify* and *Get features* commands are also extended to support *LightNVM* specific attributes like number of channels in the controller, address space per channel etc.

2.5.1 Extended I/O commands

Unlike the I/O commands of NVM Express specification where the NAND flash addresses are contiguous in a single I/O command, in LightNVM I/O command, the NAND flash addresses need not be contiguous and hence a mechanism similar to the PRP lists is provided to handle multiple NAND flash addresses per command. The I/O commands are extended to support two different implementations.

Hybrid I/O

In case of hybrid I/O, the device manages the persistency of mapping table and host offloads the consistency and durability of translation tables. Hence with every *write* command, the host sends the updated translation table entries using an approach similar to PRP lists and the device updates its local translation table. In case of a *read* the host would send only the logical addresses and device would look up the local translation table for physical addresses.

Physical I/O:

In this case, the host manages all the metadata and the persistency of translation tables. In this case, the host will only send a physical address list using a mechanism similar to PRP lists for every *read* and *write* command.

In addition to *read/write* I/O commands, an additional *erase* command is also provided.

2.6 PCI EXPRESS SPECIFICATION

PCI Express (PCIe) [13] is a high speed serial interconnect standard. It replaces the earlier parallel bus architecture of PCI, PCI-X while still maintaining software compatibility. It can no longer be called a bus. A better way is to describe it as a network tree consisting of switches, endpoints and root complex. Figure 2.9 gives an overview of PCI Express architecture.

2.6.1 Outline

The *root complex* connects the processor and memory subsystem to PCI Express switch fabric consisting of *end point* devices which can generate or consume PCI Express packets, or *switch devices*, which are responsible for extending the root complex ports and routing the packets upstream and downstream.

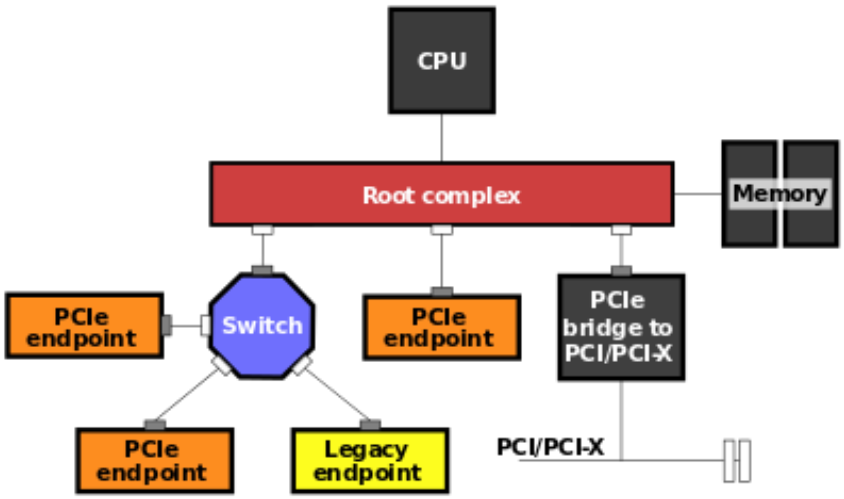

Figure 2.9: PCI Express topology [14]

Physically, PCI Express devices connect through *lanes*. Each *lane* is a full-duplex serial byte stream, transporting data packets in "byte" format simultaneously in both directions. A PCI devices can consist of one or more lanes in the powers of two. Lane counts are written with a prefix "x". A "*PCIe Gen3 4x*" device implies that the device supports Gen3 PCI Express specification and consists of 4 physical lanes. The combination of lanes between devices is called a *link.*

PCI Express uses a packet-based layered protocol just like an Ethernet protocol, consisting of transaction layer, a data link layer and a physical layer as shown in Figure 2.10. The *physical layer* consists of analog interfaces and logic circuits required for encoding and differential transmission on the *link* using the available number of *lanes*. The *data link layer* is responsible for making sure that the packets arrives correctly by adding its own headers and

Link CRC. Note that if the data link layer gets a positive acknowledgment from the upstream partner, it only means that the packet safely reached a nearby switch, no end-to-end acknowledgment is ever made in PCI Express nor is necessary. The *transaction layer* is the PCIe upper most layer and is responsible for building the packets, hereby referred to as TLP (*Transaction Layer Packet)*. In this thesis, we only work at the *transaction layer* and only the description of the same is provided in the section below. For description of other layer refer to [13].

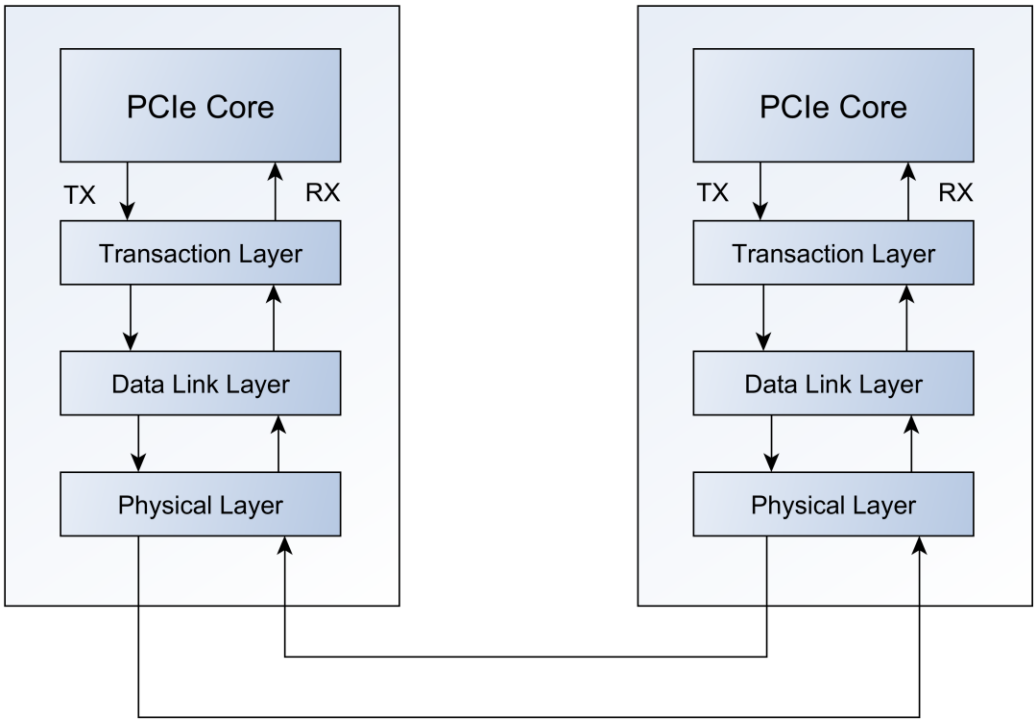

Figure 2.10: PCI Express layers

2.6.2 Transaction Layer protocol

PCIe uses a packet based protocol to exchange information between *transaction layers* of two components communicating over a *link.* Transactions are mainly carried out through *requests* and *completions.* There are mainly two kinds of requests – *read request* and a *write request.* The origin of the request can either be at the root complex or at the end point. However, for the end point to do *bus mastering i.e.*, to be an originator of a read/write request , there are two-things that needs to done. Firstly, the endpoint should be granted bus mastering by setting

the "*Bus master enable"* bit in one of the standard PCIe *configuration registers.* Secondly, the host software must program the PCIe BAR (*Base Address Register)* with an allotted physical address space to the PCIe endpoint.

2.6.2.1 Request packet

A general PCI Express write/read packet is shown in the Figure 2.11.

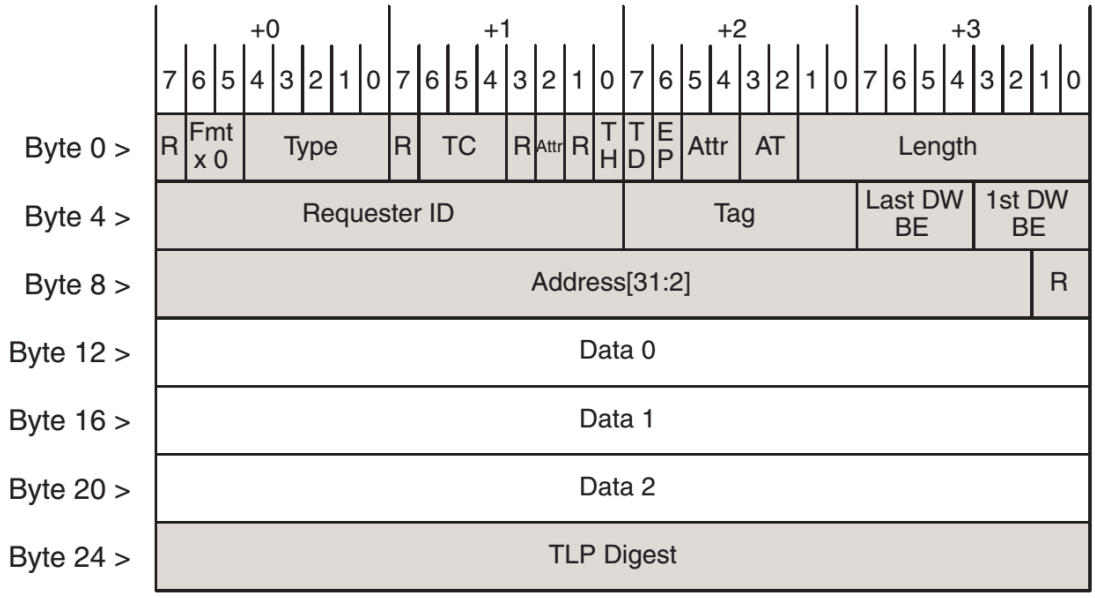

Figure 2.11 PCIe request packet format [13]

The following is the brief description of various fields in the packet.

- The *Fmt* field, together with the *Type* field determine whether it is a read request or a write request. For the *write* request, the *Fmt* is 0x2 and *type* is 0x0, for a *read* request, the *Fmt* is 0x0 and type is *0x0.*
- The *R* fields indicates that the fields are *reserved.*
- The *TC* field stands for *Traffic Class* and is used to create Virtual Channels to setup independent flow control for packets. Nevertheless, this field is almost always set to zero.

- The *Attr* field is used to set packet order rules and *no snoop* attribute. The default value of this field is also zero which implies PCI strongly ordered model and hardware enforced cache coherency.
- The TD bit is also normally set to zero. It indicates that there is no extra CRC on the *TLP Digest* field of the packet. It is usually unnecessary since the link layer has its own CRC and makes sure it is error resilient.
- The *Length* field in case of *write* indicates that the amount of data payload present in the packet (Data 0, Data 1 etc. in figure 2.12*)* in DWords. In case of *read*, it indicates the number of DWords that needs to be read starting from the address specified in the *Address* field.
- The *Requester ID* corresponds to the sender ID, this is set in PCIe configuration space registers.
- The *tag* field is simply ignored by the receiver in case of a *write* and the sender can place anything in that field. In case of *read*, tag is significant. The completer of the read copies this field into the completion TLP and this allows the requester to match the completion with its outstanding requests. Multiple outstanding requests from a single device is allowed and hence the requester *uniquely* tags all its outstanding requests to identify the responses.
- The 1st BE DW (*Byte Enable DWord)* allows to choose which of four bytes in the first data DW are valid when the length is 1 and it is zero if length is greater than 1.
- The last BE DW indicates byte enable for last DW of request. If length is 1, this field is zero, else it has to be set.
- The *Address* field is simply the address to which the first DW has to be written or read from and rest of DWords are written in contiguous address.
- The DWords of PCIe are specified in Big Endian format. Note that typical Intel processors are Little Endian.

2.6.2.2 Completion Packet

The structure of a completion TLP is shown in the Figure 2.12.

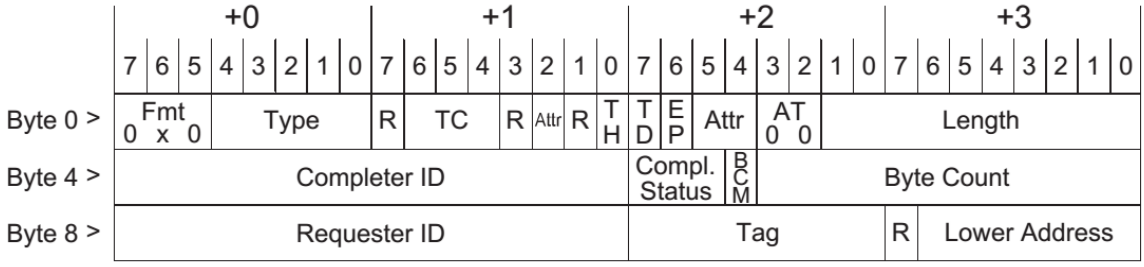

Figure 2.12: PCI Express completion packet [13]

The following is the brief description of various fields in the packet.

- The first DW is similar to that of a request packet. The *Fmt* and *Type* fields which are used to identify the completion packet are set to 0x2 and 0xA respectively.
- The *Length* field corresponds to the length of the completion. Note that the length of the completion can be less than the number of DWs requested for. When this happens, the response is split across several completions.
- The *Byte count* field indicates the number of bytes yet to be transmitted including those in the current packet. This field is useful in case of split completions.
- The *Lower address* field is the lower 7 bits of the address of the first byte of the data presented in the completion TLP. This again will be useful in case of split completions.
- The *Completer ID* corresponds to the sender ID.
- The *Requester ID* corresponds to the requester ID. It is same as the requester ID of the read request packet which requested for the completion.
- The *Status field* being zero indicates that the completion is successful, else it indicates rejection.
- The BCM field is zero, except when the packet originates from PCI-X bridge.

2.7 OTHER RELATED PROTOCOLS

The *ONFI (Open NAND Flash Interface)* [15] protocol is a standard high-speed interface specification for NAND flash devices.

RapidIO protocol [16], similar to PCI Express protocol is a high speed host interconnect protocol, however, it has much better performance and lesser transmission overhead. It has the capability to replace standard PCI Express protocol, especially in multi-processor environment or Ethernet protocols.

CHAPTER 3

ARCHITECTURAL DESIGN AND IMPLEMENTATION

This chapter briefly introduces system level architecture and high-level description of the blocks from the perspective of a storage controller. Following the description, we will look at the scope of the work presented in this thesis from this perspective and a brief introduction to *Bluespec System Verilog* in which the current controller is implemented. Then we will delve into the architectural description and implementation details of each of the storage controller blocks.

3.1 SYSTEM LEVEL ARCHITECTURE

Figure 3.1 shows a block diagram of the system level architecture.

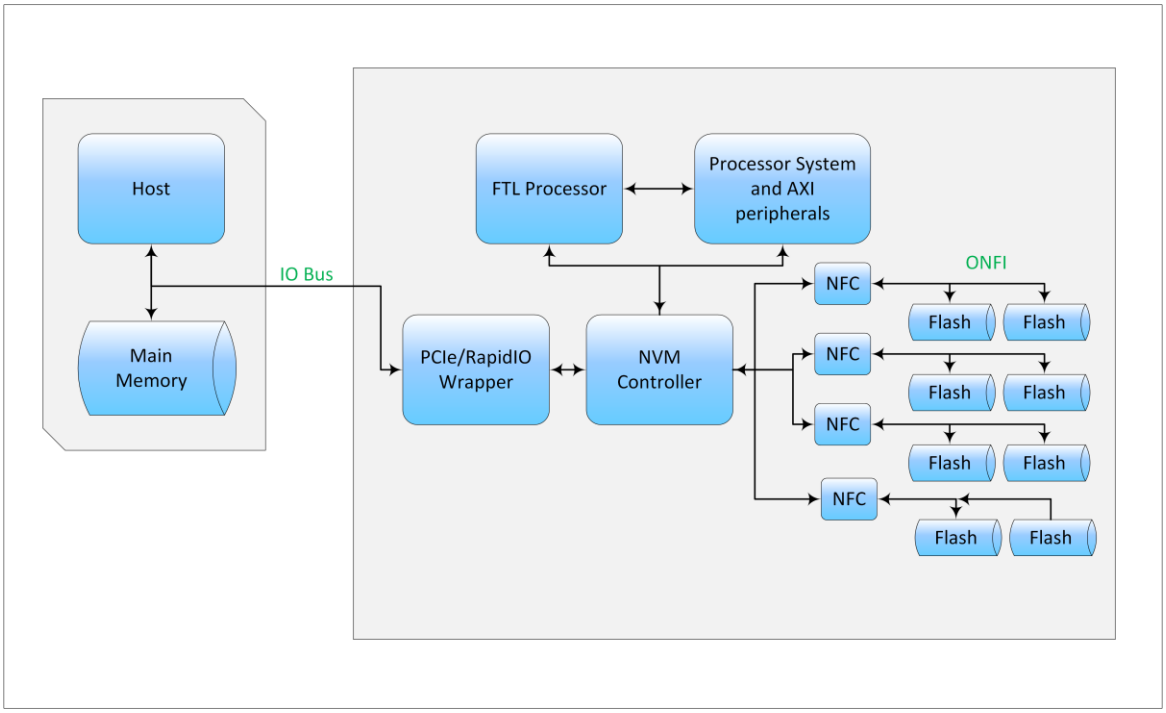

Figure 3.1: System level controller architecture

At the system level, the storage topology is divided into the following.

3.1.1. CPU/Host:

The CPU/host can be single or multi-processing entity that makes use of the underlying controller for IO accesses to a non-volatile memory. The IO accesses can be at various levels of abstraction as supported by the underlying controller which includes, host-side FTL or device-side FTL, Physical IO or Hybrid IO etc., that issues and processes commands as per the NVM Express specification. The host is mainly responsible for the following

- Software initialization of the underlying controller and programming the state of the underlying controller based on the capabilities of the controller.
- Setting up I/O submission and completion queues and submission of I/O commands to the appropriate queues based on the priority of the IO command.
- Setting up and updating queue doorbells to indicate the completion or arrival of a new IO command in the appropriate queue.
- Handling or off-loading logical to physical address translation based on the specific use case of device-side FTL or host-side FTL implementations.
- Handling or off-loading garbage collection and translation table maintenance based on capabilities of the underlying controller and per the use case.

3.1.2 Main Memory:

The main memory corresponds to the memory address space accessible for bus mastering by the controller where the host maintains IO command queues, buffers to transfer the read data from the NVM or from which data is to be transferred to the NVM and those memory pointers which store metadata and address lists associated with an IO command.

Main memory would also act as a cache for storing translation tables in case of device-side FTL implementations while the persistency of translation tables may be maintained by underlying non-volatile medium.

3.1.3 I/O interconnect:

I/O interconnect plays a very important role in the performance of the entire system. It is responsible for the transfer of data between the main memory and the controller. The latency,

overhead of the protocol, bandwidth and signaling speed of the I/O interconnect forms a crucial role in the performance of the system as whole. In addition, the topology of interconnect and original goal of interconnect design also places a major role in interoperability of the system. For eg: *PCI Express* is designed to connect peripheral devices to a host processor and has a tree-based structure with a central root complex and switches connecting the nodes flowing down the tree. Because of this topology, PCI Express natively doesn't support peer-to-peer host connectivity and using such an I/O interconnect in a distributed processing system or storage provides a challenge. *RapidIO,* on the other hand, was designed to work as a fabric interconnect enabling ease of peer-to-peer communications and better routability making it an interesting choice for storage systems, however lack the advantage of being natively present in the traditional motherboards and hence the extra cost.

3.1.4 NVM Controller:

NVM Controller forms a bridge between the bus controller and the NAND flash controller. It is mainly responsible for the following.

- Providing host configuration and control registers.
- Fetching, execution and dispatch of IO commands issued by the host into the IO queues.
- Providing a generic interface for integration with any peripheral bus/fabric controller (eg: PCI Express/ RapidIO) and providing a generic host interface and device FTL interface for host-side and device-side FTL implementations respectively.
- Pipelined execution and parallelization of IO commands across channels, deciding the IO bandwidth and channel utilization.
- Coalescing the response on completion of an IO command and returning the status to host.

3.1.5 PCIe/RapidIO controller:

The PCIe / RapidIO Controller provides a bridge between the *NVM Controller* generic host interface and transaction interface of the PCIe / RapidIO core. It validates the transaction packets received from the host, dispatches them across the generic NVM Controller interfaces, constructs bus specific transaction packets/signals from the payload responses generated by

NVM Controller interfaces and provides appropriate wait logic for resource conflicts from generic NVM interfaces to bus/fabric interfaces.

3.1.6 FTL Processor:

FTL Processor on the device has use case specific roles depending on whether the implementation use host-side or device-side FTL.

In case of host-side FTL implementations, since the translation is already done on the host-side, it merely acts as a meta-data processor and scheduler. It interprets the meta-data provided from the host IO commands, generates and schedules NAND flash commands across the channels based on the physical block address of the issued IO command.

In case of device-side FTL implementations, FTL modules takes in the IO commands consisting of main memory page address, logical block address and length of IO request and returns back scheduled NAND commands to NVM controller channels consisting of physical block addresses, main memory address and length post-translation.

3.1.7 NAND Flash controller:

NAND flash controller provides a bridge between NVM Controller generic NAND interface and ONFI interface of NAND flash chips. Every channel has an independent NAND flash controller and each NAND flash controller can control one or more NAND chips. It executes NAND commands issued by the NVM Controller and returns the status of the NAND command to the NVM controller. It also implements error correction for reliable NAND data reads, and also maintains persistent bad block tables and returns them to the NVM Controller when requested for.

3.2 SCOPE OF WORK

This thesis deals with the architectural design and implementation of generic mutli-channel NVM Controller, integration of NVM Controller with PCI Express bus interface and testing the SOC through FPGA emulation. The PCI Express is chosen for all its advantages and because of its ready availability on any traditional mother board and Xilinx FPGA. For the purposes of this thesis, the NAND flash controller and the underlying NAND flash are emulated using BRAMs in the simulation and BRAMs/DDR3 in FPGA emulation. The focus

of the current thesis is on architectural features of the NVM Controller to design a highly scalable, low latency, high throughput storage controller while providing generic interfaces for interoperability and extensibility. A comprehensive simulation environment with automatic test case generator and configurable speed NAND flash model are also designed for functional and performance testing. It also prototypes functional testing of the controller through FGPA emulation and host-side linux kernel module designed specifically for testing purposes. It also attempts to prove the potential of the designed controller architecture through simulations and evaluations.

3.3 BLUESPEC SYSTEM VERILOG

The complete design of blocks and most of the testing environment w.r.t this thesis is written in *Bluespec System Verilog (BSV).* It has been chosen for its advantages and some of them are described here.

BSV raises the abstraction level of hardware, especially the control logic when compared to the traditional hardware description languages like *Verilog* and *VHDL*. It models hardware using *atomic rules* which fire based on *explicit* and *implicit* conditions. The *explicit* conditions are user-defined and *implicit* conditions are inferred from the *methods* used inside the rules. *BSV* models interfaces between blocks using *methods* which has *implicit* enable and ready signals. Hence, whenever the method of a module is invoked, in other words, whenever an input is provided or output from the module is used in the *rule,* the *implicit* conditions of the method are automatically applied to the rule firing conditions. In this model, the conditions of when an input or an output of a module can be accessed can be encoded in *method* conditions of that module and the *Bluespec* compiler will make sure those conditions are met whenever we use those module instantiations. This particular feature of *Bluespec* makes the design of control logic much easier and reduces the development time drastically. It also makes the reasoning of correctness easier since designer only needs to think in terms of step-by-step execution of rules and the rule conditions.

In addition, *BSV* offers highly parametrisable constructs, overloaded interfaces and functions which enables the designer in developing generic and scalable hardware. This feature makes the design highly modular and hence extremely maintainable which is very useful especially when designing generic hardware like the one in this thesis. It has powerful

static type checking which removes huge amounts of potential bugs during the stage of compilation itself.

It also provides pre-defined library elements like FIFOs, BRAMs etc. which are modelled using *BSV* methods. This makes the modelling of control logic in pipelined designs using FIFOs and buffers much easier. The code written in *BSV* will also be compact because of all the above described features increasing the flexibility of design.

Despite its advantages, it also has some disadvantages in terms of area and timing of the generated logic. A more controlled implementation of the logic in Verilog/VHDL would result in better area and timing. Also, interfacing with modules written in Verilog/VHDL will be difficult because of different interface modelling abstraction of both these languages.

The rest of the chapter is focused on design and implementation and is divided into three sections:

- Design and Implementation of NVM Controller
- Design and Implementation of PCI Express wrapper
- Design and Implementation of round robin arbiter.

3.4 DESIGN AND IMPLEMENTATION OF NVM CONTROLLER

The *NVM Controller* implements the host protocol specified in Section 2.4 and some of features in Section 2.5. The architecture is an extended version of work presented in [17]. However, it only forms as a basis for the current architecture with significant changes and extended features. To enable modularity and performance, some of the state machines and architectural features are re-designed, while some remained the same. However, for the sake of completeness, some of the state machines that are un-changed are also presented briefly in this chapter.

The top level data flow of the NVM Controller is shown in Figure 3.2. The commands that are handled by the controller are divided into four parts:

- Commands that change the state of the controller and does not require data transfer. (*Execute Command).* All other commands are *dispatched* to respective state machines.

- Commands that request controller capabilities/features and requires data transfer. (*Data Structure Command Execution).*
- Commands that are targeted towards the underlying physical NAND flash. (*Data Transfer Command Execution)*
- Commands that are processed by the additional plugins to the controller. (Out of band commands, not implemented in the current version).

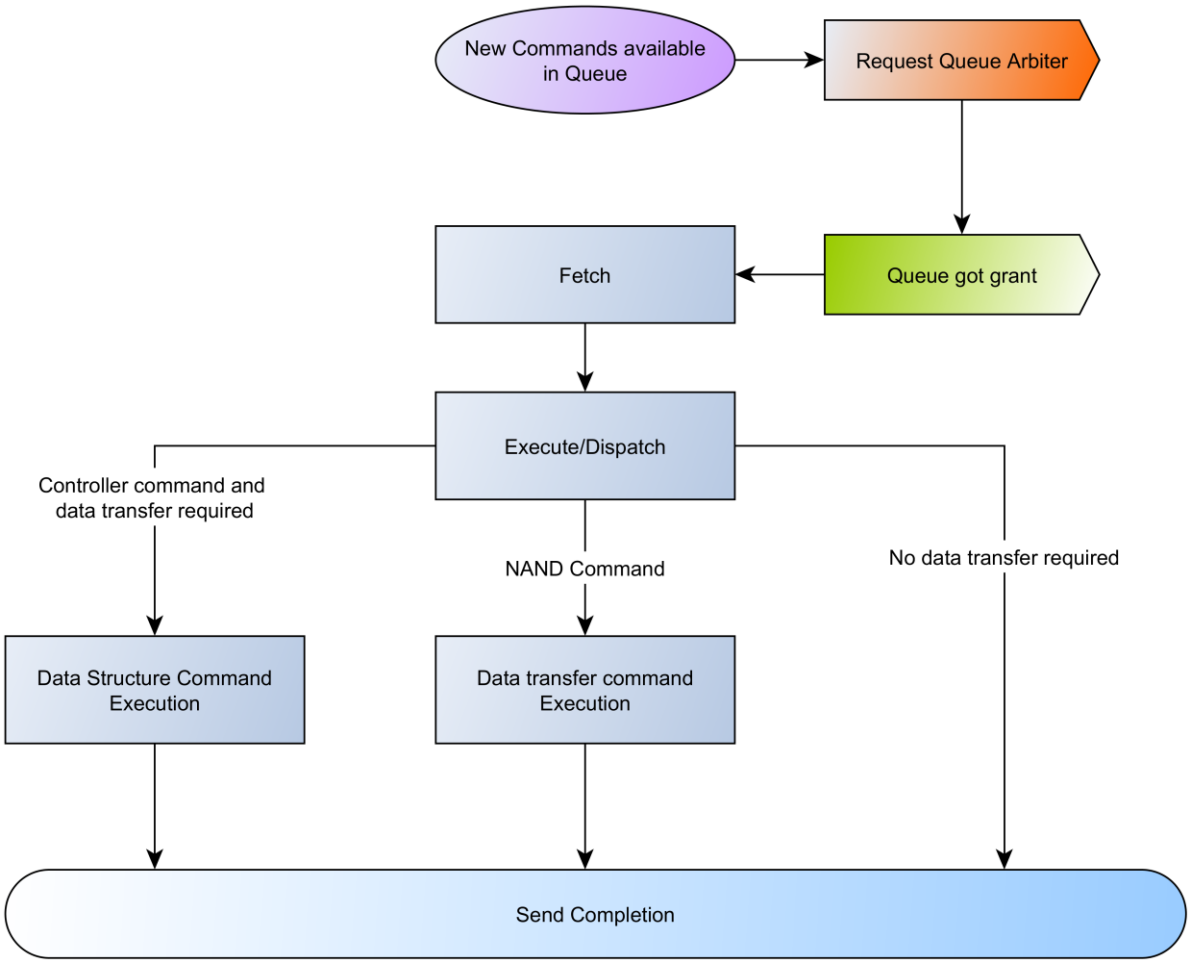

Figure 3.2: Top level data flow of NVM Controller

Each of the above commands go through independent data paths and are executed in parallel. However since the responses to the host, that are to be sent, should go through a single IO interface (PCI Express in this case), all those paths which require access to the host IO interface will request a PCIe arbiter and the arbiter allocates grants in a round-robin

fashion. Each executed command generates a Completion to the host which is handled by *Completion State Machine.*

3.4.1 Interfaces to NVM Controller:

The NVM Controller has the following interfaces.

Configuration Interface:

The configuration interface provides a memory mapped host control registers. It consists of a *read_* method and a *_write* method. All the access are either Dword aligned or native width accesses as per the NVM Express specifications.

IO Transmit Interface:

The Read and Write requests by the NVM Controller to the main memory goes through this interface. This interface is similar to an AXI4 Interface. It has following methods:

- *write_* method returns 1 if the request is a write and 0 if it is a read.
- *data_* method returns the data to be written if the request is a write request else it is invalid. The data width can be set as 32/64/128 or higher using WDC parameter.
- *address_* denotes the main memory address to which the data is to be transferred to or requested from depending on whether it is a write request or a read request respectively.
- *payload_length_* denotes the size of data to be transferred or being requested in DWords to or from the main memory.
- *data_valid_* indicates if the data returned by other methods is valid or invalid.
- *tag_* method represents a unique ID for each outstanding read request. This tag has to be returned to the controller along with the read completion which is used by the controller to identify which IO client does the completion data belongs to.
- *wait_* method, when called, indicates the controller that the IO interface is not ready and hence the controller retains the current state of the methods. A single strobe of data transfer is complete when *data_valid_* is asserted 1 and *wait_* method is asserted 0 simultaneously.

Interrupt Inteface:

The interrupt interface generate an interrupt to the host when a new command completion has been sent to the main memory. It has two methods: *vector_rdy* method which indicates that there is an interrupt and *vector_number* which indicates the interrupt vector.

Completion Interface:

All the completions for read requests sent by the NVM Controller to main memory comes through this interface along with appropriate *tag_* corresponding to that request. The data width of the completion is same as the data width used for transmission and is set by WDC parameter.

NAND flash interface:

This interface is used for all command transactions with the underlying NAND flash. If the NVM Controller wants to perform a read request on the NAND flash, the request is sent through *request_address_* method, similarly if it has to perform an erase request, the request is sent through *request_erase_address_* method. If the NAND flash is ready with the data corresponding to a read request, then a *_interrupt* is to be generated by the NAND flash and the data is taken in through *_data_in* method when controller is ready. If the controller wants to perform a write operation on NAND flash, the address and data is provided to the NAND flash through *data_out_* method and once the NAND flash completes the write, the status of write is returned through *write_status_* method.

FTL Processor interface:

This interface is used to communicate with the FTL processor module. All the I/O commands consisting of *opcode, logical block address, length, main memory address* and *metadata* are sent through *ifc_ftl_processor_in* interface and the scheduled NAND commands consisting of *opcode, physical block address, length* and *main memory address* are to be sent by the FTL processor to the NVM Controller through *ifc_ftl_processor_out* interface. The PRP addresses corresponding to read commands and write commands are to be sent separately through *put_prp_read* and *put_prp_write* methods. Separate methods are added in order to facilitate priority execution in the later versions.

The number of *NAND flash interfaces*, *ftl_processor_out* interfaces and the corresponding channel logic scales automatically with the NO_CHANNELS parameter.

3.4.2: Fetch State Machine:

Figure 3.3 shows the data flow of the Fetch State machine. The *Fetch state machine* is responsible for fetching the commands en-queued by the host in IO queues, present in main memory, into the controller for execution.

IDLE

If any of the *Submission Queues* available is enabled and there are IO commands yet to be processed i.e. the queue is non-empty, then each of those *Submission Queues* requests the *Submission Queue Arbiter*, which is round-robin arbiter, for access to the fetch state machine. Once one of the *Submission Queue* gets the grant, the fetch state machine switches its state from *IDLE* to *FETCHING_COMMAND* state.

FETCHING_COMMAND

In *FETCHING_COMMAND* state, it requests the IO interface arbiter (PCIe arbiter in this case) for access to the host IO interface and once access is obtained, a read request is sent to the corresponding command address of granted submission queue with a fixed request tag for fetch state machine, and the fetch state switches to *WAIT_FOR_COMMAND* state. The command address is calculated from the *submission queue base address*, which is stored locally on *Create Submission Queue* command, and the *Submission Queue head pointer.*

WAIT_FOR_COMMAND

Once the completion with tag corresponding to the fetch state machine is received through the controller's completion interface, the command is en-queued into *internal execution queue* for execution, the corresponding *Submission Queue head pointer* is incremented and fetch state switches back to *IDLE***.**

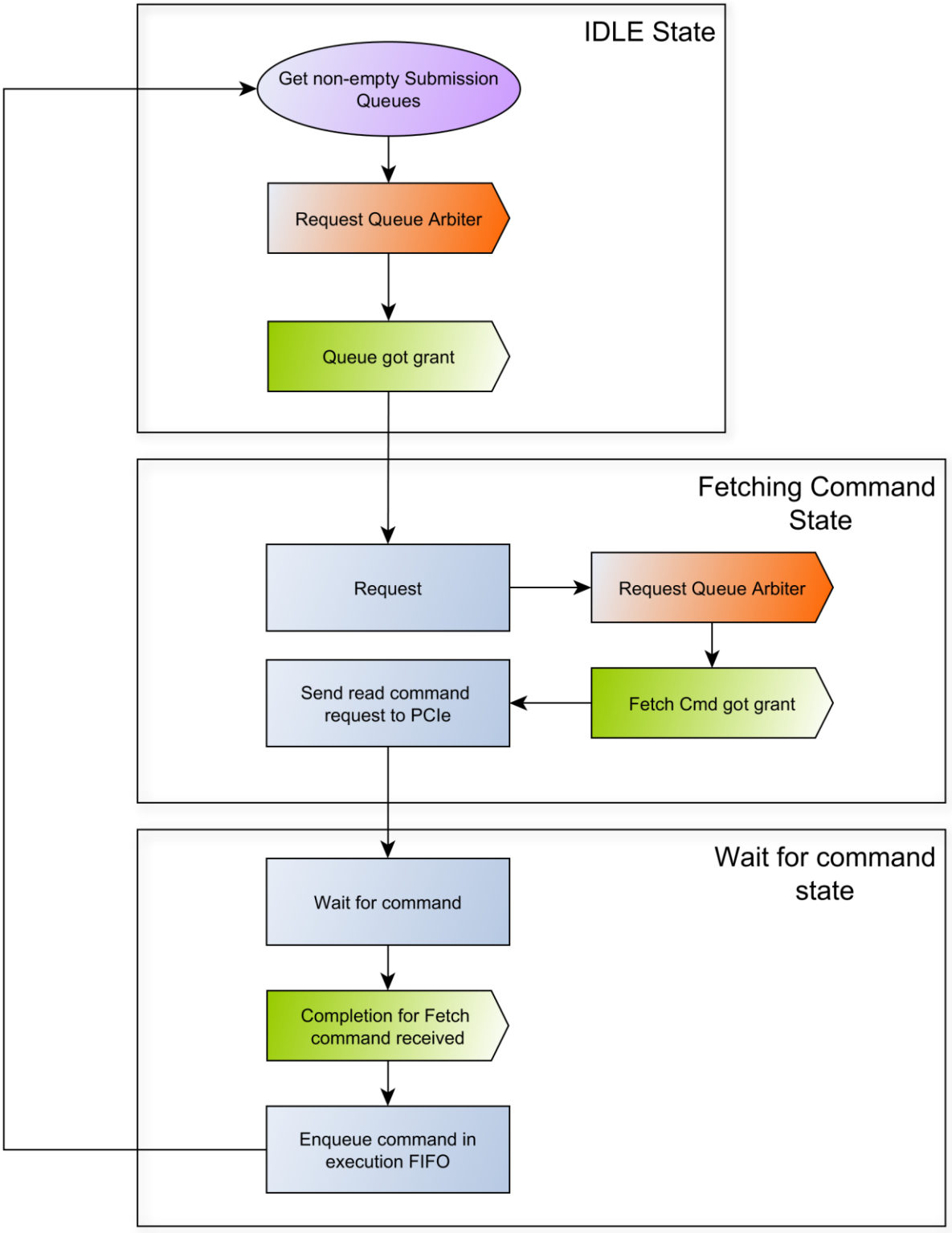

Figure 3.3: NVM Controller fetch state machine

3.4.3 Execute/Dispatch state machine:

Figure 3.4 shows the data flow of Execute/Dispatch state machine. The commands that are to be processed by the controller are divided into four types as specified in the introduction of Section 3.3. It is the function of the Execute/Dispatch state machine to either execute those commands which doesn't require any data transfer and takes only single cycle to complete (or) dispatch the commands to appropriate state machines like *Data Structure Command Execution* and *Data Transfer Command Execution*.

IDLE

If there are commands present in the *Internal Execution Queue*, en-queued by the *fetch state machine*, *execute state machine* switches from *IDLE* state to one of the execute states depending on the type of command. If the command comes from the submission queue with ID zero, then it is an admin command and the state switches to *EXECUTE_ASQ_COMMAND* else it is an NVM Command and the state switches to *ISQ_CHECK_ABORT***.**

EXECUTE_ASQ_COMMAND

The supported commands that are executed in *EXECUTE_ASQ_COMMAND* state includes *Create/Delete IO queues*, *Identify Command*, *Abort Command* and *Set/Get features* command.

On execution of *Create/Delete IO Queue* command, the corresponding IO Queue is enabled or disabled and the base address of the IO queue is written into the IO base address register and IO Queue head is initialized to zero. The base address register and IO queue head are used in calculation of address of the command to be fetched in *fetch state machine*.

Identify Command returns an identify data structure as required by the NVM Express specification and since it requires a data transfer, this command is dispatched to *Data Structure state machine*.

Abort Command adds the ID of an IO command submitted previously by the host, into one of the submission queues, to the *Abort List*. However, aborting the command would only be successful, if the corresponding IO command is not yet processed by *Execute State machine.*

Get/Set features either returns/sets the capabilities of the controller and hence changing the state of the controller based on the NVM Express specification.

ISQ_CHECK_ABORT

The *LightNVM I/O Commands* that are supported are *Read NAND, Write NAND* and *Erase NAND*. All these are commands are to be checked for a possible abort against the *Abort List* in *ISQ_CHECK_ABORT* before proceeding to *EXECUTE_ISQ* state.

EXECUTE_ISQ

In execute ISQ state, the I/O commands are dispatched to *Data Transfer Command Queue.*

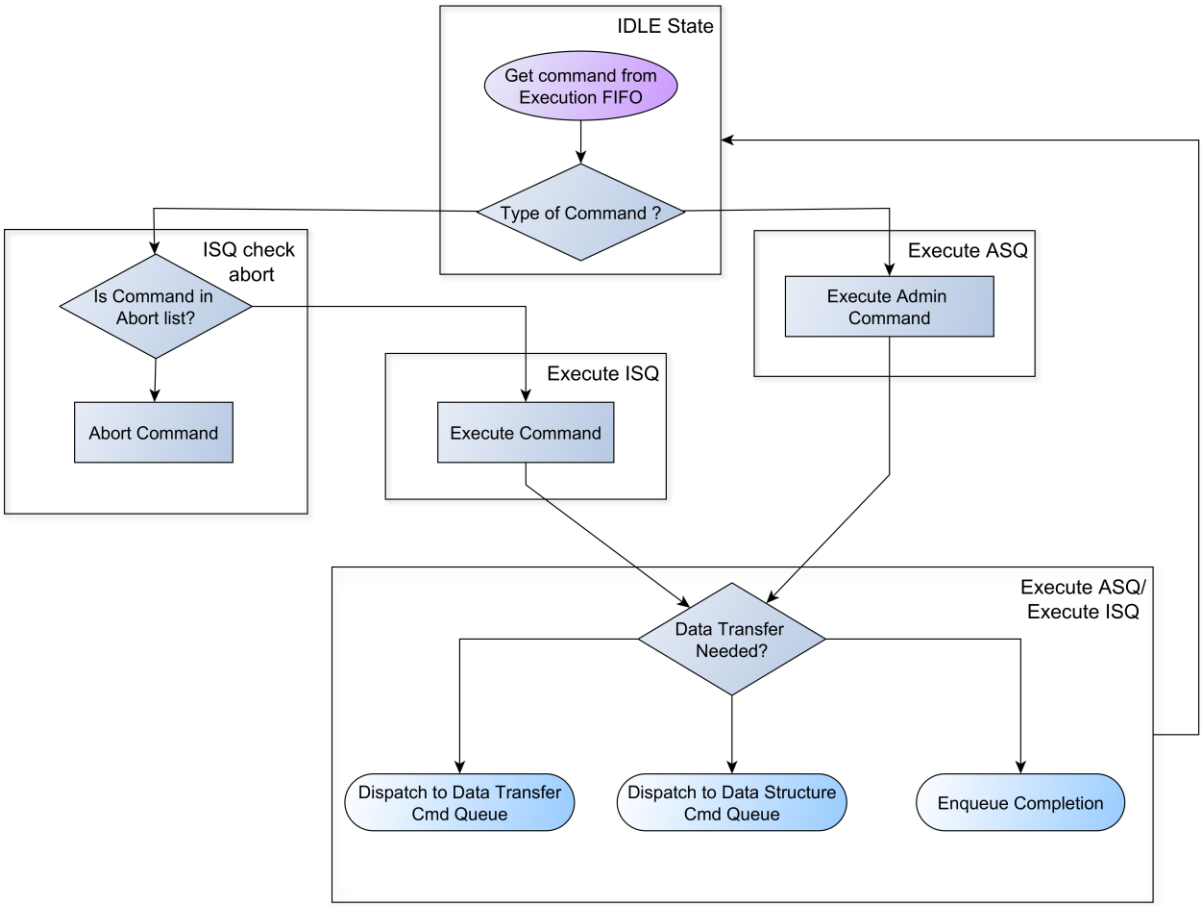

Figure 3.4: NVM Controller execute state machine

3.4.4: Data Transfer State Machine:

The *Data Transfer State Machine* processes commands en-queued by the *Execute Command* state in the *Data Transfer Command Queue.* Since all these commands are targeted towards

an underlying NAND flash, the state machine is so designed to incorporate genericity of FTL processing, extensibility of NAND interfaces while still maintaining performance. The high level block diagram of the *Data Transfer State Machine* is shown in Figure 3.5.

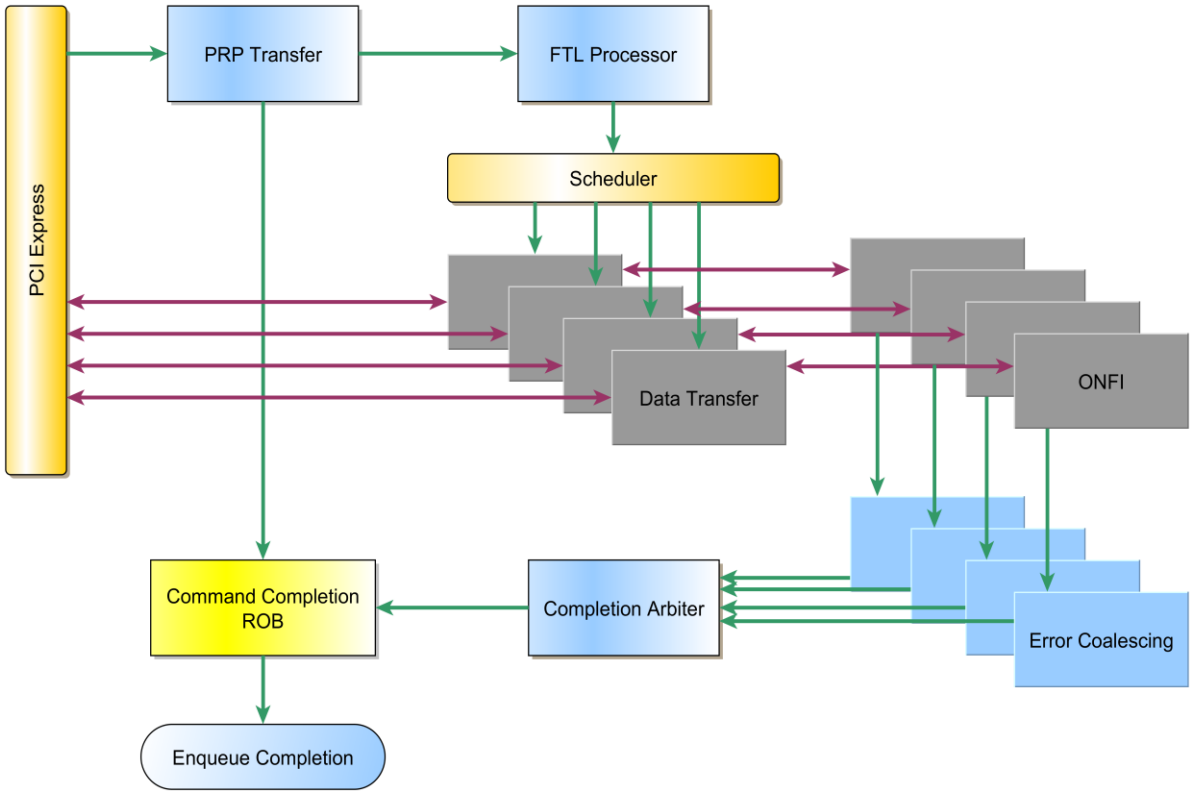

Figure 3.5: Block diagram of NVM controller data transfer state machine

This state machine is further divided into the following four parts operating independently and in a pipeline.

- PRP Transfer
- FTL Processing/Scheduling
- Data Transfer
- Status Coalescing

3.4.4.1 PRP Transfer:

The *PRP Transfer* logic is responsible for fetching the list of main memory addresses and the list of NAND flash addresses corresponding to the pages that needs to be written to (or) read

from and send them to the FTL processor. In case of device-side FTL implementations, no NAND flash address list needs to be fetched since the IO requests point to contiguous NAND addresses based on NVM Express Specification, however, in case of host-side FTL implementations since the translation is done on the host-side, NAND physical address list is also needed for IO command processing. Depending on whether the controller implements hybrid IO or physical IO (see section 2.5), the one to one logical addresses and physical addresses is to be received or physical address alone has to be received respectively.

To handle all these three cases while maintaining the use case specific changes to minimum, the PRP transfer state machine categorizes the transfers into two parts, *PRP TRANSFER (main memory address list)* which is common for all the implementations and *Metadata transfer*, which corresponds to physical address list in case of physical IO host-side implementations, one to one physical and hybrid address list in case of hybrid IO host-side implementations and ignored in case of device-side FTL implementations. The *METADATA TRANSFER* state has to be modified appropriately based on the use case, however this modification is minimal and only requires routing the data to different FTL interfaces. The following lines of this section discusses *host-side physical IO* use case which is currently implemented and then presents how the state machine is to be modified for the other two use cases.

Figure 3.6 shows the state flow of PRP transfer state machine.

IDLE

If the data transfer command queue, which is en-queued by the *Execute state machine* is non-empty, PRP transfer state machine sends the IO command to the FTL processor, stores the command attributes to the *Command completion reorder buffer (CCROB)* and switches its state from *IDLE* state to *SEND_PRP1* state. The IO command is tagged with the address of the CC-ROB which needs to be returned back along with the scheduled command post-FTL and is used in status coalescing.

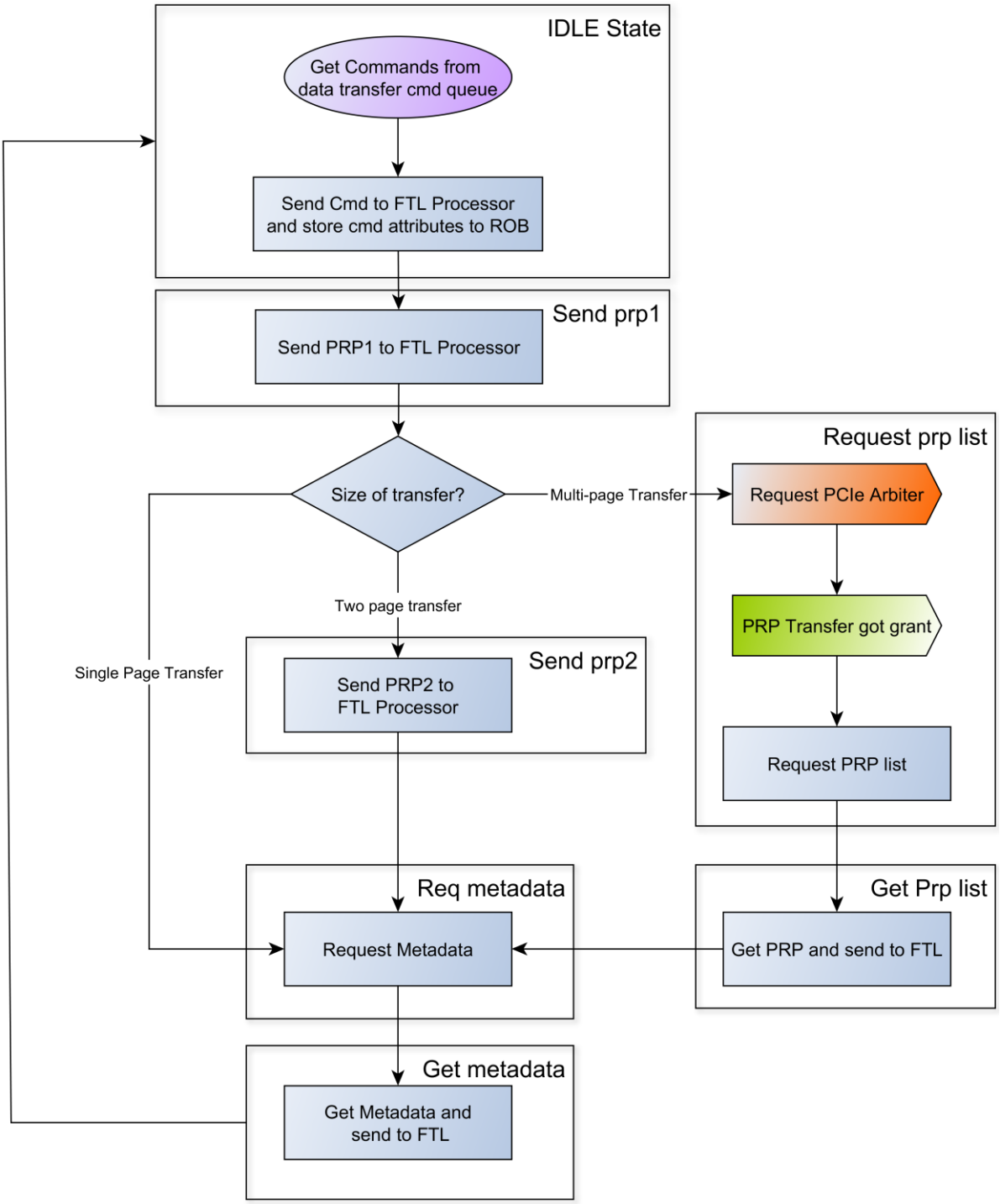

Figure 3.6: NVM Controller PRP transfer state machine

SEND_PRP1

In *SEND_PRP1* state, the PRP1 address is send to the FTL processor, the buffer space for the storing the PRP lists of the outstanding commands has to be provided inside the FTL processor with the current implementation. However, this can be moved into the CC-ROB and PRP lists can be fetched only when required i.e., post-FTL scheduling of the write command and post-completion for a read command. In case of read and write commands, PRP1 address corresponds to first main memory address of data transfer. In case of erase command, PRP1 address corresponds to the NAND physical block address (and not main memory address) that needs to be erased and FTL processor needs to interpret it so.

The main memory address for the data transfer are specified within the IO command for single page and two-page transfer, however for multi-page transfer, a PRP list is used. (See section 2.4.5.1). So, depending on the size of transfer, the *PRP transfer state* switches to *SEND_PRP2,* if it is a two-page transfer (or) *REQUEST_PRP_LIST* state if it is a multi-page transfer (or) *REQ_METADATA* state if it is a single page transfer

SEND_PRP2

In *SEND_PRP2* state, the PRP2 address is sent to the FTL processor and the state switches to *GET_METADATA* state.

REQUEST_PRP_LIST

In case of *REQUEST_PRP_LIST* state, the PCIe arbiter is requested for access and once the grant is obtained, it sends a request for PRP list fetch and switches to *GET_PRP_LIST* state.

GET_PRP_LIST

In *GET_PRP_LIST* state, the state machine waits until a completion corresponding to the *PRP transfer state machine* has been received and once the completion is received, the PRP list is forwarded to FTL processor. On completion of the PRP list transfer, it switches to *REQ_METADATA* state.

REQ_METADATA and GET_METADATA

The *REQ_METADATA* and the *GET_METADATA* states are similar to that of the PRP list states and once the metadata is obtained, it is sent to the FTL processor and the state machine returns to *IDLE* state.

In the case of Physical IO host-side FTL, the metadata corresponds to a list of physical address to which the data is to be written to or read from the NAND flash and current implementation corresponds to this. However, in case of hybrid IO host-side FTL, the metadata corresponds to one to one logical and physical block address and hence has to be sent along two different FTL interfaces, one corresponding to logical block address and physical block address. In case of device-side FTL, the *metadata* depends on the use-case. If there is no metadata, the metadata transfer states can be skipped altogether.

3.4.4.2 FTL Processing/ Scheduling:

For host-side FTL implementations, FTL processor merely acts as meta-data interpreter and scheduler. It receives the *metadata* from the PRP transfer state of the controller and provides it to the *scheduler*. The *scheduler* evicts the NAND commands with physical addresses to the *Data Transfer logic* corresponding to the appropriate channel of the controller. The FTL module, in this case, also provides buffer storage for the PRP lists and physical address list.

In case of device-side FTL implementations, the FTL processor receives the IO commands from controller, does the page translation to be implemented in a software stack residing on the on-board processor and presents the mapped commands to the scheduler. The *Scheduler* evicts the mapped commands to corresponding IO channels of the NVM Controller. The logic for the device-side FTL algorithm is not in the scope of this thesis, only the logic for handling host-side FTL, interfaces for handling device-side FTL and scheduler is implemented in this thesis.

The physical address presented to the scheduler is divided to two parts: *1) Channel address* and *2) local address.* The no. of bits of the physical address corresponding to the local address is parametrized based on the NAND_MEM_SIZE, which corresponds to the size of the NAND flash per channel. This requires that the size of the NAND flash channel to be aligned to *Channel address*, else the continuity of the address space is lost and hence is not desirable. This restriction is not a limitation since most of the commercial NAND chips comes with sizes in powers of two and hence are implicitly *Channel address aligned*. This also reduces the complexity of scheduler significantly when compared to using unaligned addresses. The one-hot encoding of the *Channel address* selects the channel to which the post-FTL commands are to be dispatched as shown in the Figure 3.7

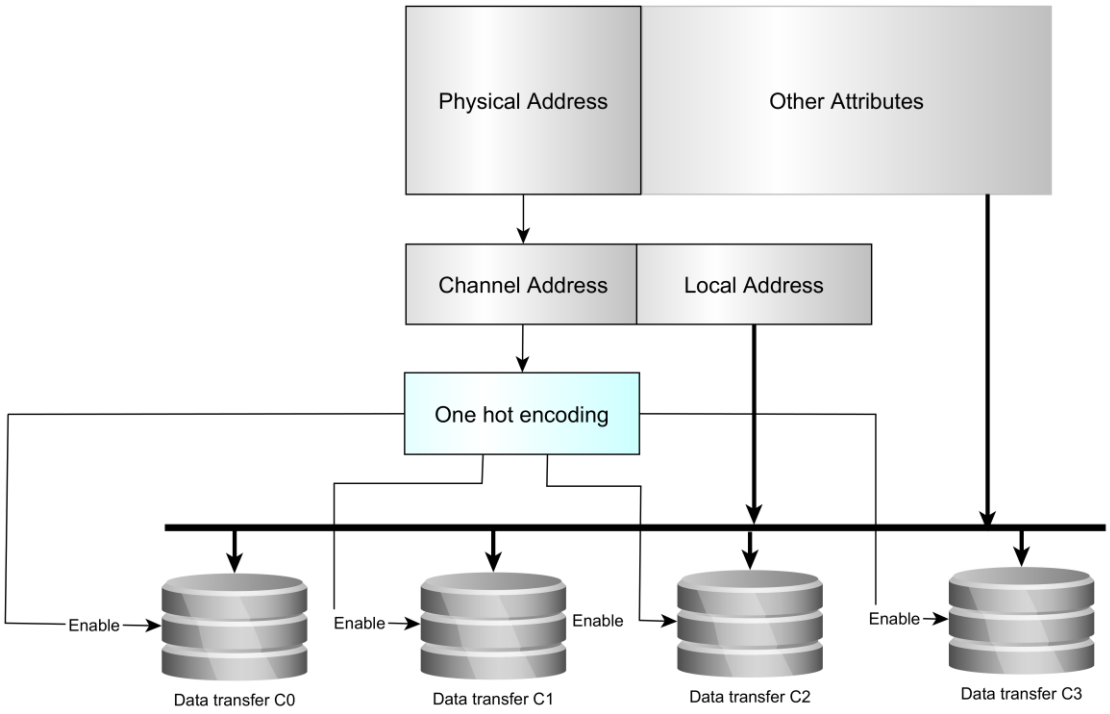

Figure 3.7 NVM Controller channel scheduler

3.4.4.3 Data Transfer

The *Data Transfer* logic is replicated across each channel and the replication is parametrized based on NO_CHANNELS parameter. The *data transfer* logic interacts with the NAND flash interface of the controller and evicts the scheduled commands to the NAND flash channel controller. In addition it also fetches the data from the main memory corresponding to a WRITE command or transfers the data to the main memory in case of a READ command.

The *Data Transfer* logic is divided into two pipeline stages as shown in Figure 3.8, *Command issue logic* and *Command completion logic.*

Command Issue Logic:

Command Issue logic dispatches the scheduled instructions and required attributes into the appropriate NAND flash interface. The description of NAND flash interface is provided in section 3.4.1.

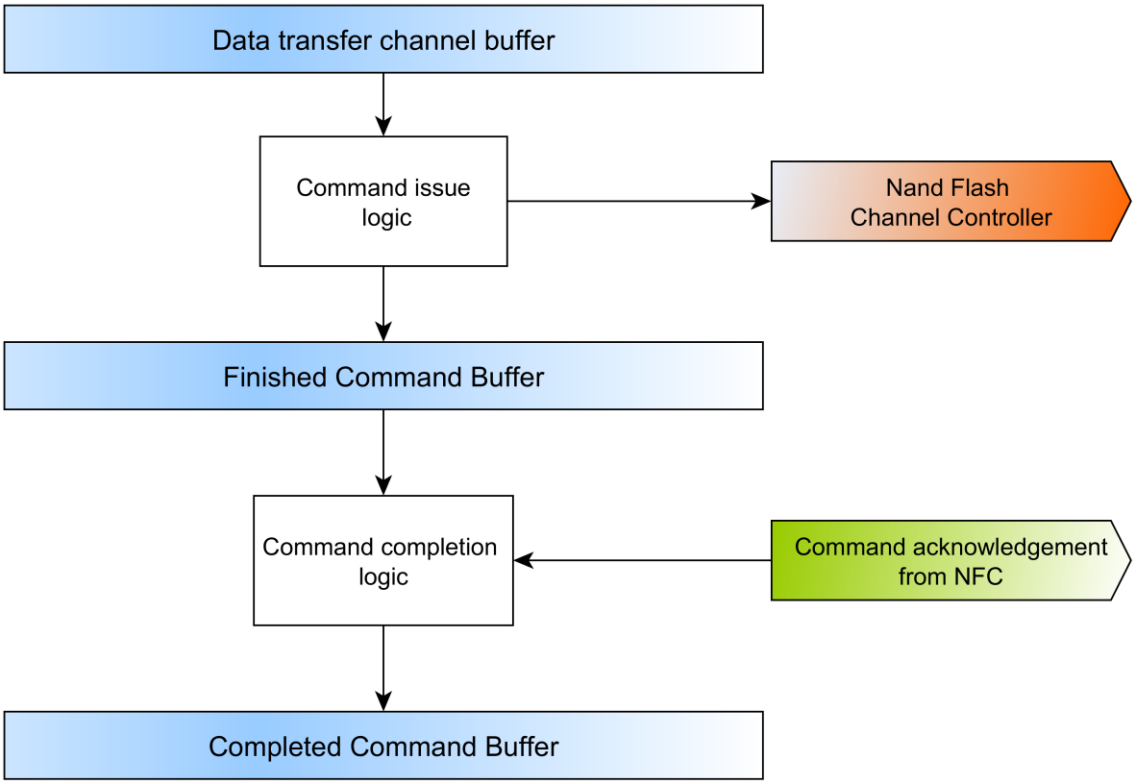

Figure 3.8: NVM Controller pipelined data transfer logic

Figure 3.9 gives an overview of command issue logic and the following lines describes it.

- If the command is a READ or ERASE command, the commands are directly dispatched to *request_address_* and *request_erase_address* interfaces of the NAND flash channel.
- If the command is a WRITE command, the data to be written also needs to be fetched from the main memory. For this, the PCIe arbiter is requested and once the grant is obtained, a read request is issued to the PCIe from the PRP address associated with the command. Recall that PRP addresses are sent to the FTL processor and FTL processor provides the buffer storage for PRP addresses. The FTL processor returns the PRP address and tag along with the scheduled command.
- Once the data is obtained from PCIe, the data is transferred through *data_out_* interface.

- Once the READ/WRITE/ERASE command is dispatched, they are enqueued into *Finished command buffer* for command completion post-acknowledgment from the NAND flash channel controller.
- Since the completion of a READ command requires PRP address to which the read data needs to be sent, these PRP addresses returned by FTL post-scheduling are stored in a FIFO and used during *command completion stage*. Note that this forces the completion for READ commands to arrive in-order, however, out-of-order completion for READ commands can be made possible by replacing the FIFO with buffer similar to Command Completion ROB or by storing PRP addresses in CC-ROB (as suggested in Section 6.1.4) and exposing the command tag to NAND flash channel controller.
- The completion of WRITE/ERASE command doesn't require any additional information and hence only the command tag is stored in the *finished command buffer*.

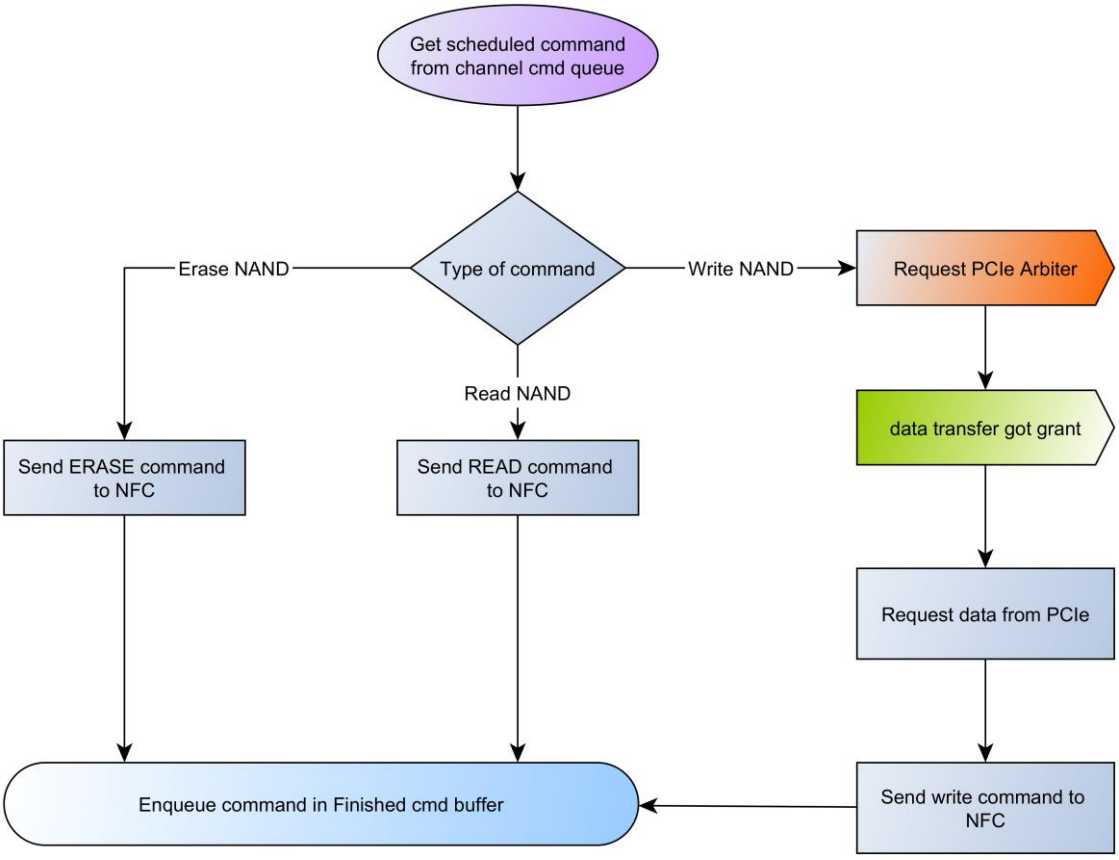

Figure 3.9: NVM Controller data transfer state machine command issue logic

Command Completion Logic:

The *Command completion logic* completes the data transfer command processing on reception of the response from NAND flash channel controller. Figure 3.10 gives an overview of command completion logic. The following lines describes the logic.

- If the command is a write NAND or erase NAND, the status of the command is tagged to the command and it is moved from *finished command buffer* to *completed command buffer*.
- If the command is a read NAND command, since the data from the NAND is to be written to the main memory, the PCIe arbiter is requested. Once the request is granted, the NAND interface is enabled and the data from the NAND interface is transferred to PCIe using the PRP address stored in *Command Issue* stage.
- Once the data transfer is complete, the read NAND command is moved from *finished command buffer* to *completed command buffer* with the status tagged.

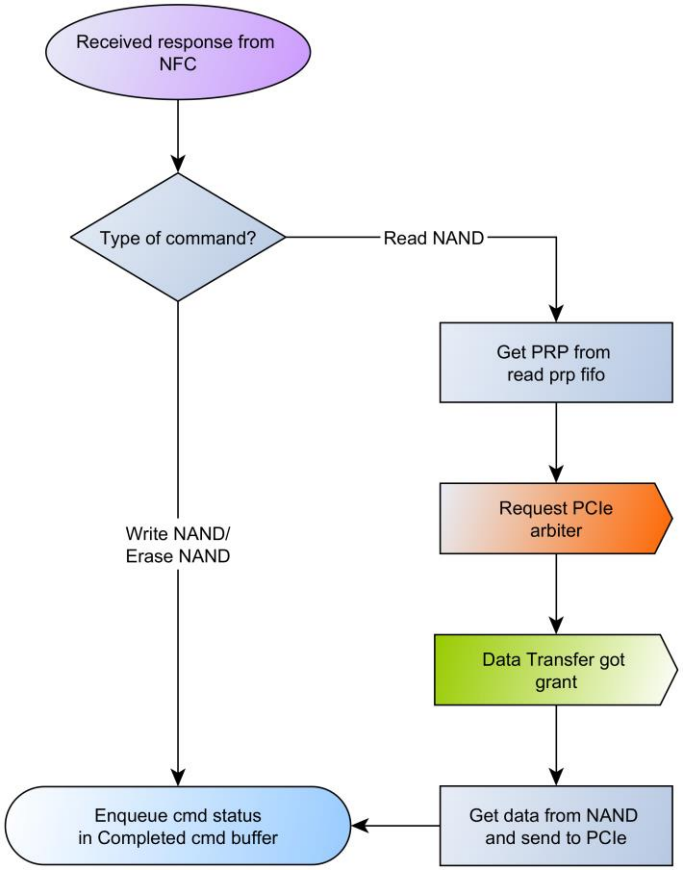

Figure 3.10: NVM Controller data transfer state machine command completion logic

3.4.4.4 Status coalescing

The *Status Coalescing* logic takes the *completed commands* processed by the *data transfer logic* and reports back to the CC-ROB where the command attributes are stored at the beginning of *data transfer state machine.* Once the entire IO request is completed, the completions are sent in-order to the host. This is however not a requirement of the NVM Express specification, but this reduces the complexity of logic of CC-ROB because it can be implemented as a simple circular buffer. It is to be noted that the buffer size of CC-ROB should be chosen carefully based on the no. of outstanding requests to NFC that are possible at a given time and number of outstanding commands in the pipeline. Since completions are committed in order, large IOs can block completion of smaller IOs following it. Out-of-order completion can also be permitted by implementing the CC-ROB as a set of registers and a queue consisting of a list of free registers and is left as an implementation choice or future work.

A single status entry per command is currently implemented. However, this will be inefficient for larger IOs since the entire IO needs to be re-tried by the host in case of a failure. For host-side FTL implementations, it would be useful to send a single status bit per every page, since it would reduce the overhead of retry while simultaneously identifying the bad-blocks. But a single status bit per page of an IO would require the maximum number of pages per IO command be limited. Since this feature is not been added to the specification yet, currently the implementation is left with single bit per entire IO. However since the status coalescing from NAND interface and CC-ROB updates are done at the granularity of page, adding single status bit per page would be straight forward. The implementation of the status coalescing in shown in Figure 3.11 and described in the following lines.

- If the *Completed command buffer* of each channel is not empty, they request command completion round robin arbiter and one of them gets the grant.
- The command tag and page tag (page tag is not currently used since only single status bit per IO is implemented else page tag should also be used) is used to update the entry of CC-ROB and number of pending acknowledgments is reduced by 1.
- The command tag corresponds to address of CC-ROB with which all commands in pipeline are tagged in *PRP Transfer state machine,* the page tag corresponds to the

page no. within the IO command and the pending acknowledgements corresponds to no. of pages yet to be acknowledged.

- Once the pending acknowledgments at the head of CC-ROB becomes zero, the command is en-queued in the completion queue which is taken on by *Completion state machine*, and the head is updated.

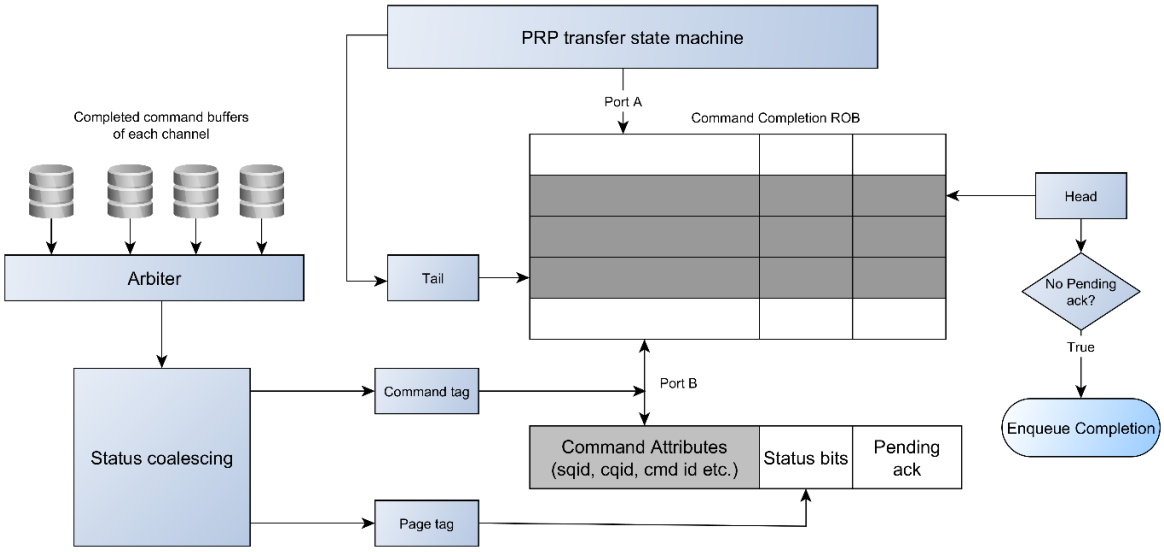

Figure 3.11: NVM controller data transfer state machine status coalescing

3.4.5 Data Structure State Machine:

The *Data structure state machine* processes commands en-queued by the *Execute state machine* in *data structure command queue*. These commands only transfer pre-defined data structures to the host which holds information about the controller, namespaces or special configurations. Figure 3.12 shows an overview of data structure state machine and the following lines describes it.

- When an entry is present in the *data structure command queue*, the state is switched from *IDLE* state to *TRANSMIT* state.
- In *TRANSMIT* state, based on the type of command *Identify namespace, Identify Controller* and *LBA range type*, the data structure pre-defined in BRAMs is

transmitted to PCIe following the same procedure as any other PCIe client in the controller by first requesting and send it when grant is obtained.

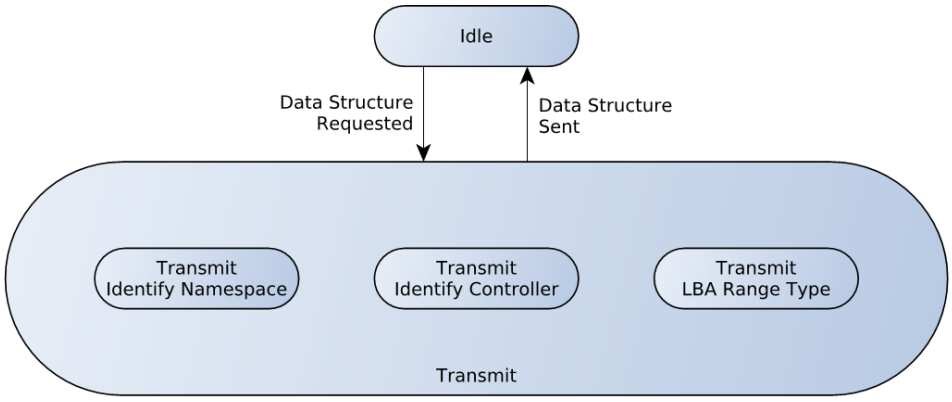

Figure 3.12 NVM Controller data structure state machine [17]

3.4.6 Completion State Machine:

The *Completion state machine* is responsible for sending the command completions using the info en-queued by various state machines including *data transfer state machine*, *data structure state machine*, *fetch state machine* and *execute state machine* in their respective completion queues. The general data flow of *completion state machine* is shown in Figure 3.13 and described below.

- When an entry is present in any of completion queues, the completion state machine switches its state from *IDLE* to *SEND* state.
- The *SEND* state requests the PCIe access and sends the completion to the appropriate completion queue based on, the completion queue id corresponding to the submission queue id of the completed command, and updates the *tail pointer* of the corresponding completion queue. It also generates an interrupt to the host.
- The interrupt masks can be controlled by the host via controller configuration interface. The host can also set an *Aggregation threshold* for each of the submission queues except the *Admin submission queue* and can receive an *aggregated interrupt* when the *single interrupts* exceeds the set threshold.

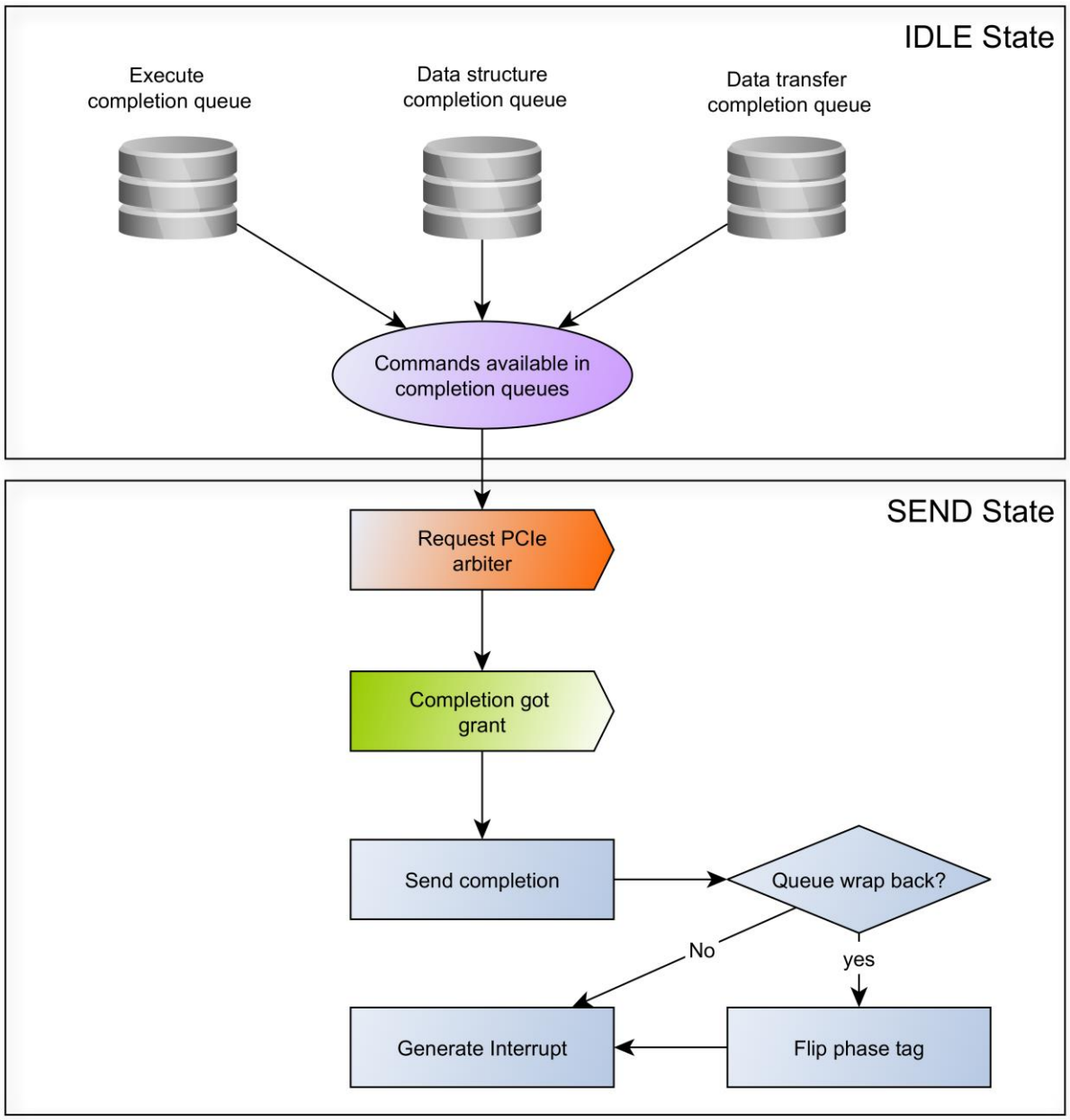

Figure 3.13: NVM Controller completion state machine

3.5 DESIGN AND IMPLEMENTATION OF PCIE WRAPPER

The *PCI Express wrapper* translates the generic NVM Controller interface into PCI Express packets for communication with the host. The PCI Express packets are presented on an AXI4 stream interface compatible with Xilinx 7 series integrated PCI Express core [18]. The main functions of the *PCI Express Wrapper* include

- Detecting PCI Express TLPs sent by PCI Express core and transmitting it to corresponding NVM Express interface
- Conflict resolution for simultaneous requests of PCI Express interface
- Convert requests from NVM express generic interface to PCI Express TLP requests.
- Split requests which exceed maximum payload length of PCI Express.

3.5.1 Interfaces to PCI Express wrapper

The PCI Express wrapper has three interfaces: *receive interface, transmit interface* and *config space interface.* The *receive* and *transmit* interfaces are standard full-duplex AXI4 stream interfaces.

Receive interface:

The TLPs (*Transaction Level Packets)* sent by the host are to be presented by the PCI Express bus controller (Xilinx 7 series PCIe core in this case) on the 128-bit *rx_in_tdata* bus of the receive interface. The signal *rx_in_tvalid* indicates the validity of data on *rx_in_tdata* bus. The signal *_rx_out_tready* indicates that the PCI Express wrapper is ready to receive the TLP. The simultaneous assertion of *_tvalid* and *_rx_out_tready* indicates that the current data on the *_rx_in_tdata* bus is processed by the PCI Express Wrapper and the core can present the next set of data on the next clock cycle. The signal *_rx_new_packet_assert* and *_rx_end_packet_assert* should indicate the start and begin of the TLP.

The receive interface also allows for straddled packets that is a new packet transmission can start in the same cycle in which another packet ends if the packet that ends only occupies the lowest significant Qword of the 128-bit interface. This is allowed in order to prevent wastage of bandwidth due to misalignment.

Transmit interface:

The TLPs (*Transaction Level Packets)* that are to be sent to the host by the PCIe Wrapper are presented to the PCIe bus controller on the 128-bit *_tx_data_out* bus. The *_tx_out_tvalid* indicates that the data presented on *_tx_data_out* is valid and the acceptance of the data is assumed by simultaneous assertion of *_tx_in_tready* signal by the bus controller. The signal *_tx_out_eof_assert* indicates the end of packet and *_tx_out_byte_enable* indicates the valid bytes within the 128-bit data stream. The bytes in 128-bit data stream will be contiguous, the *_tx_out_byte_enable* indicates invalid bytes only if they are at the end of the packet.

The *PCI Express wrapper* mainly consists of two state machines *receive state machine*, which handles the TLP presented on *receive interface* and routes them to the appropriate interfaces of NVM Controller, and *transmit state machine*, which translates the NVM controller transmit signals into TLPs and presents them to PCI Express bus controller

3.5.2 Receive State Machine

The data flow of receive state machine is shown in Figure 3.14. From the NVM controller perspective, there are three kinds of TLPs that can be received by the controller.

- TLPs that write to NVM controller configuration interface.
- TLPs that request a read from NVM controller configuration interface.
- Completions from the host as a response to an earlier read request by the NVM controller. These responses go to the completion interface of NVM controller.

The flow is described in the following lines.

- Initially, the state machines get triggered into *DECIDE_INTERFACE* state when a valid TLP is available on the receive interface and when new packet signal is asserted.
- The type of the packet is decided based on the *format* and *type* fields of a PCI Express packet (see 2.6.2 Transaction Layer protocol). If it is a configuration read packet, since a response needs to be sent to the host, the packet is enqueued into the *config_read* FIFO which is handled by *transmit state machine.* If it is a configuration write packet, the data is written into the configuration write interface of NVM Controller. If it is a completion packet, the state machine switches to *COMPLETION* state.

- The length of the completion received is determined by the length field of PCI express packet and the state machine stays in *COMPLETION* state until the entire completion is received and transmitted to NVM controller and switches back to *DECIDE_INTERFACE* state. If a new packet is asserted at the end of the packet, i.e., a straddled packet then the partial data is stored and switches to *CONFIG_READ* or *CONFIG_WRITE* or *COMPLETION* state where the rest of data is received and executed.

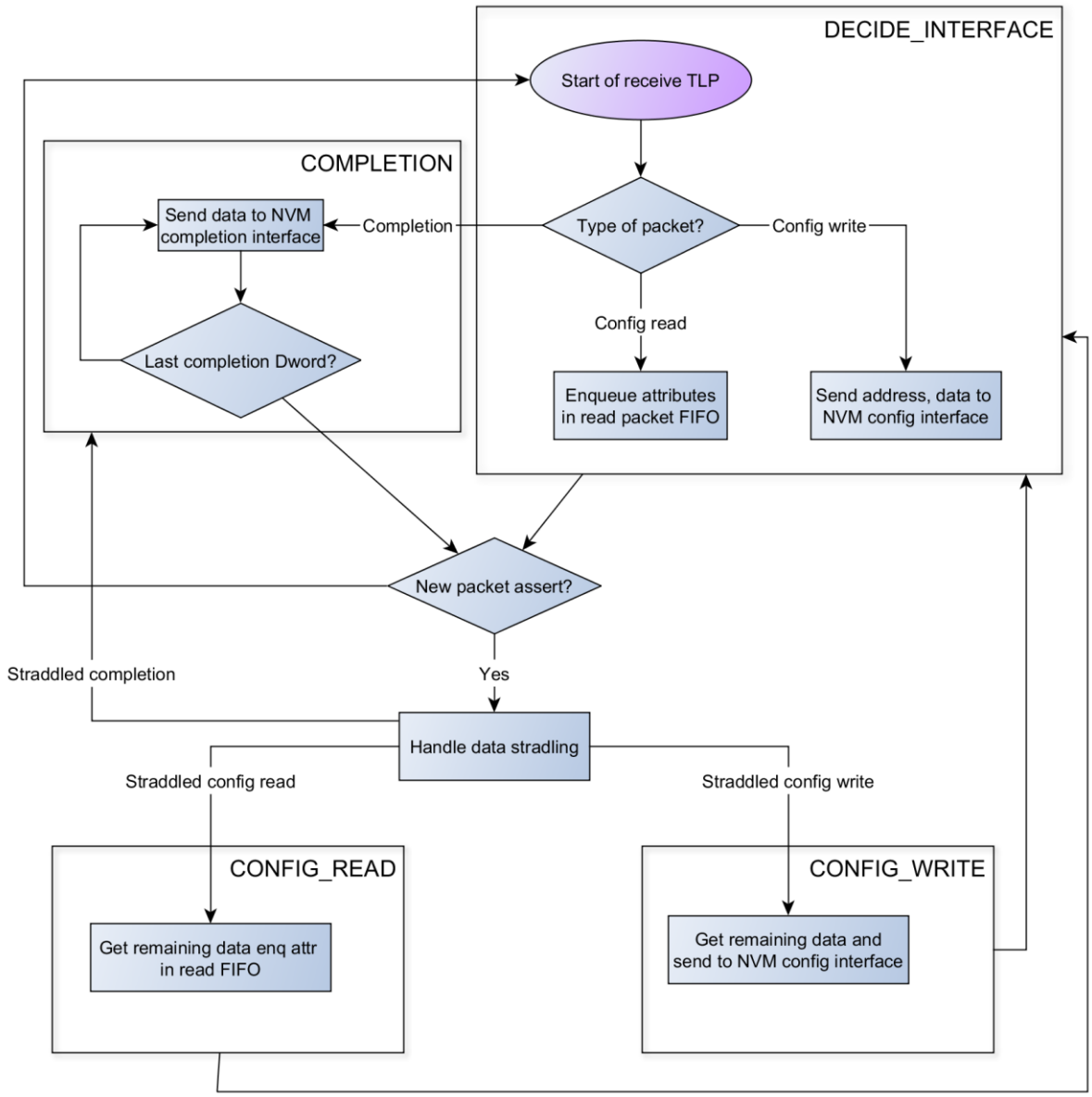

Figure 3.14: PCI Express wrapper receive state machine

3.5.3 Transmit State Machine

The data flow of *transmit state machine* is shown in Figure 3.15. There are three types of packets that needs to be transmitted to the host by *transmit state machine*.

- Write request with payload to main memory from NVM controller.
- Read request to main memory from NVM Controller
- Completion to an earlier read request by host which is stored in config read FIFO by *receive state machine*

The flow is described in the following lines

- There are two clients for *transmit state machine.* One is from the pending responses for earlier config read requests enqueued by *receive state machine* in config read FIFO, the other is the request from NVM Controller.
- If config read request is available, then a wait signal is issued to NVM controller *pcie interface* to hold NVM controller pcie requests until *config read* requests is processed and the state machine switches to *MRC_TRANSFER (Memory Read Completion)* state
- If data request from the NVM controller is available a header is sent to pcie based on the type of data request, read or write and the state switches to *DATA_TRANSFER*.
- If both requests are available, the config read request is given priority.
- In *MRC_TRANSFER* state a memory read completion packet is created and sent to PCI core.
- In *DATA_TRANSFER* state depending on the payload length of the request, the packet is sent as is or is split across multiple transactions. The maximum payload length of a PCIe transaction can be set using MAX_PAYLOAD_LENGTH parameter.

3.6 DESIGN AND IMPLEMENTATION OF ROUND ROBIN ARBITER

The round-robin arbiter is one of the components of NVM Controller. However it is presented here in a separate section because the architecture of round-robin arbiter plays an important role in determining the timing of the NVM controller especially, the I/O queue arbiter which resides in the critical path as the number of I/O queues increases beyond 64.

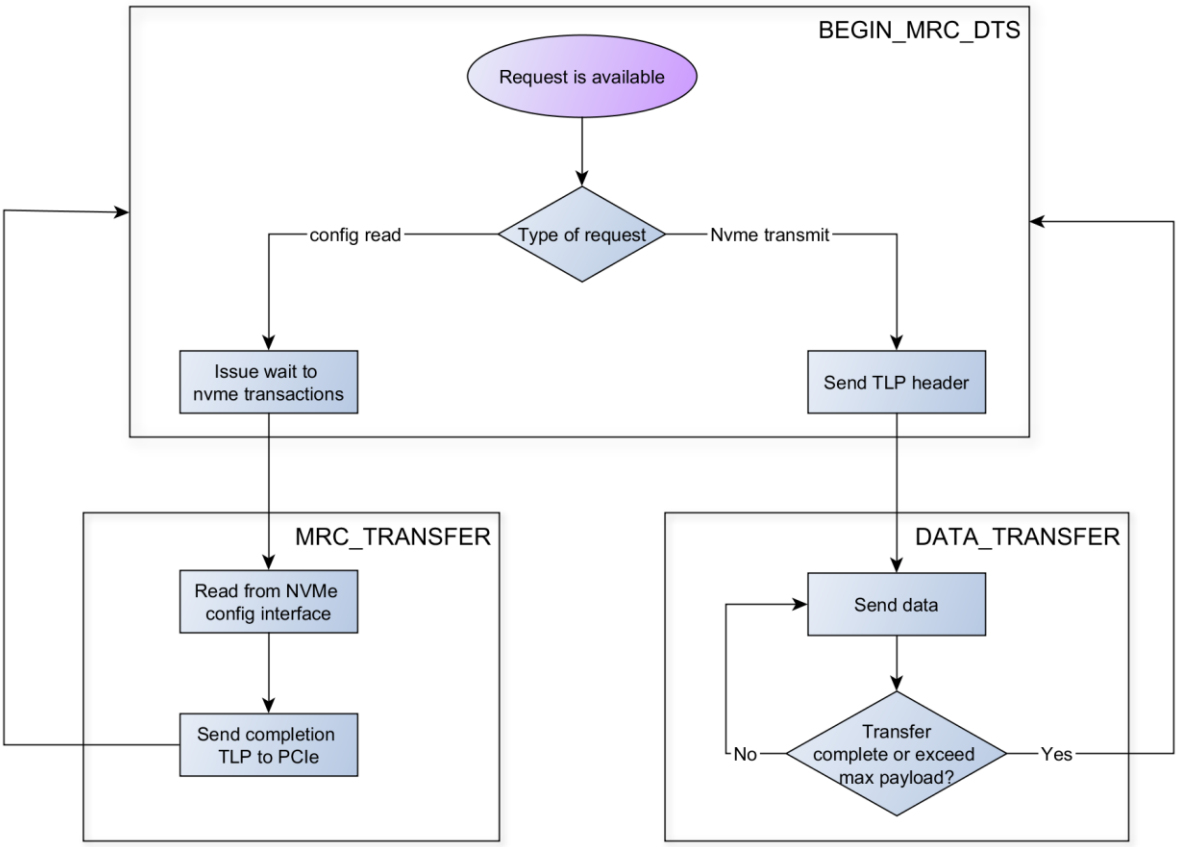

Figure 3.15: PCI Express transmit state machine

Round robin arbiter exists as a library element in Bluespec, however the round-robin arbiter in Bluespec library has *O(n)* delay where *n* is the number of clients and quickly hits the critical path. The round-robin arbiter implemented in this section has *O(log n)* delay, parametrized and the operation is divided into stages which can easily be pipelined, if needed.

3.6.1 Logic description

The general architecture of the round robin arbiter is shown in the Figure 3.16

Let N be the length the request vector R which is the input to the round robin arbiter. The value of the M^{th} element of the request vector R denoted by R_M is 1 if the client M requests for the grant else it is zero. Let G be the grant vector and the P^{th} element of the grant vector G_p is 1 if the P_{th} element gets the grant else it is zero and it also obeys that only one or none of the elements gets a grant in a given cycle.

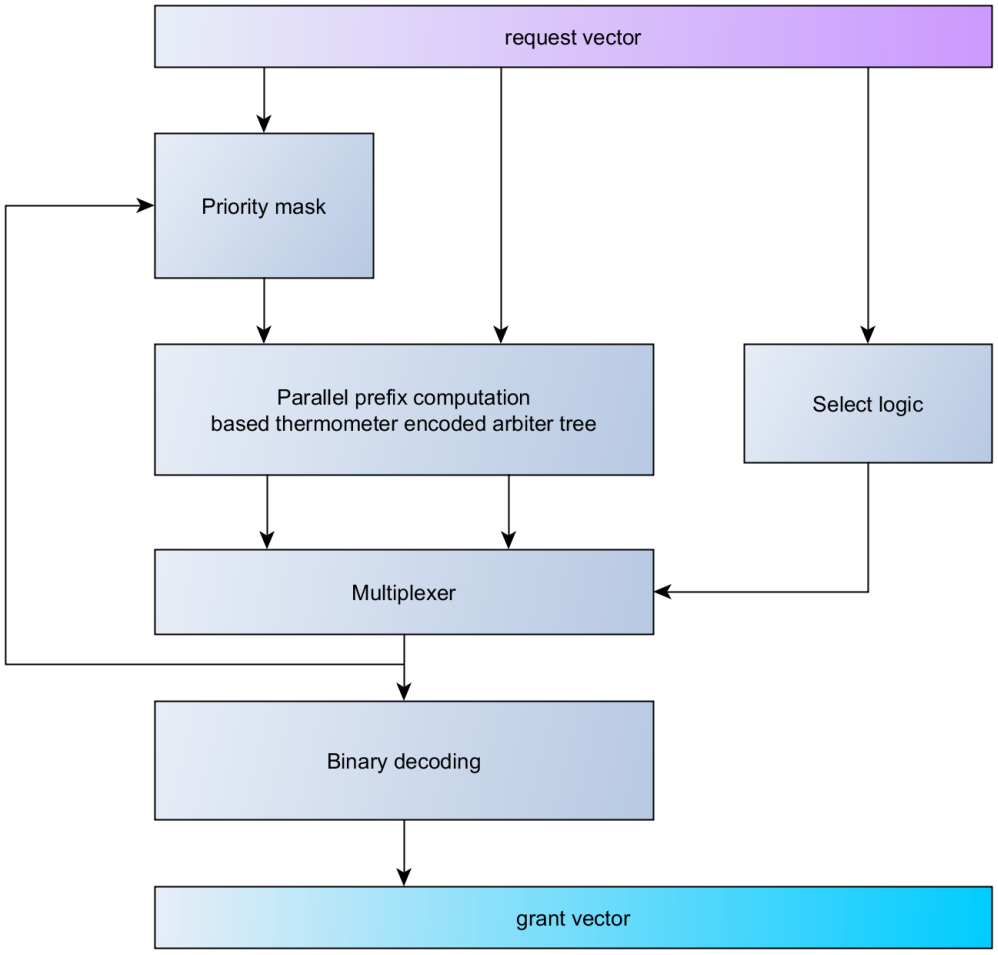

Figure 3.16: Architecture of round-robin arbiter

Now the priority of round-robin arbiter is defined in the following way. If client *M* got the grant in the previous cycle, then the priority is highest for the client next to *M* in the right-ward sense of rotation and decreases as one goes towards the right. Let *f (G)* denote a thermometer encoding of grant vector G. If P^{th} client got the grant and hence the value of G_P is 1, then $V = f(G)$ is defined as $V_K = 1$ for all $K > P$ else 0. For eg. the thermometer encoding of grant vector 0001000 is 0001111.

Now we will need to achieve a wrap-around priority functionality in order to implement a round-robin arbiter. This is achieved by considering two vectors. Let us take the request vector masked with the thermometer encoding of the previous grant vector and call it the masked vector. Now if the masked vector is non-zero, i.e., at least one of the clients following the

previously granted client is a requester, then passing the masked vector through a find-first-set based priority logic will give the grant vector. If the masked vector is zero then passing the unmasked request vector through find-first-set based priority logic would give the grant vector.

Therefore, the *priority mask* in Figure 3.16 corresponds to thermometer encoding the previous grant vector and select logic corresponds to determining if the masked vector is zero. "*find-first-set"* based priority logic is implemented using "*OR parallel prefix computation tree*".

3.6.2 Parallel prefix computation OR tree

The "*parallel prefix computation OR tree"* is shown in the Figure 3.17. It enables the arbitration logic to find quickly the first request which is asserted, resulting in the ability to find the next valid requestor with the minimal possible logic cones. It also has the advantage that the output of this OR PPC logic is a thermometer encoded vector of the current grant vector which will enable us to calculate the next masked vector without additional logic. The next mask would be the current thermometer encoded vector right shifted by one bit. The grant vector can be obtained from the thermometer encoding by the shifting the vector by one bit towards the right and exclusive ORing with itself. This corresponds to *binary decoding* in Figure 3.16.

The PPC tree itself has to be parametrized. For this purpose, the wires at the output of each level of PPC are divided into groups and group size at level '*l'* is given by 2^l starting with $l = 0$. If the number of clients is *N*, number of levels is given by *ceil (log (N)).* The advancement from one level to next happens by "OR"in every odd group with the last element of previous even group as shown in Figure 3.17.

Note that all the PPC tree does is to OR every element of the vector with all the previous elements. By doing this, all the elements following the first element with value 1, will also be made one which is the thermometer encoding of the required result and *binary decoding* will give the required result.

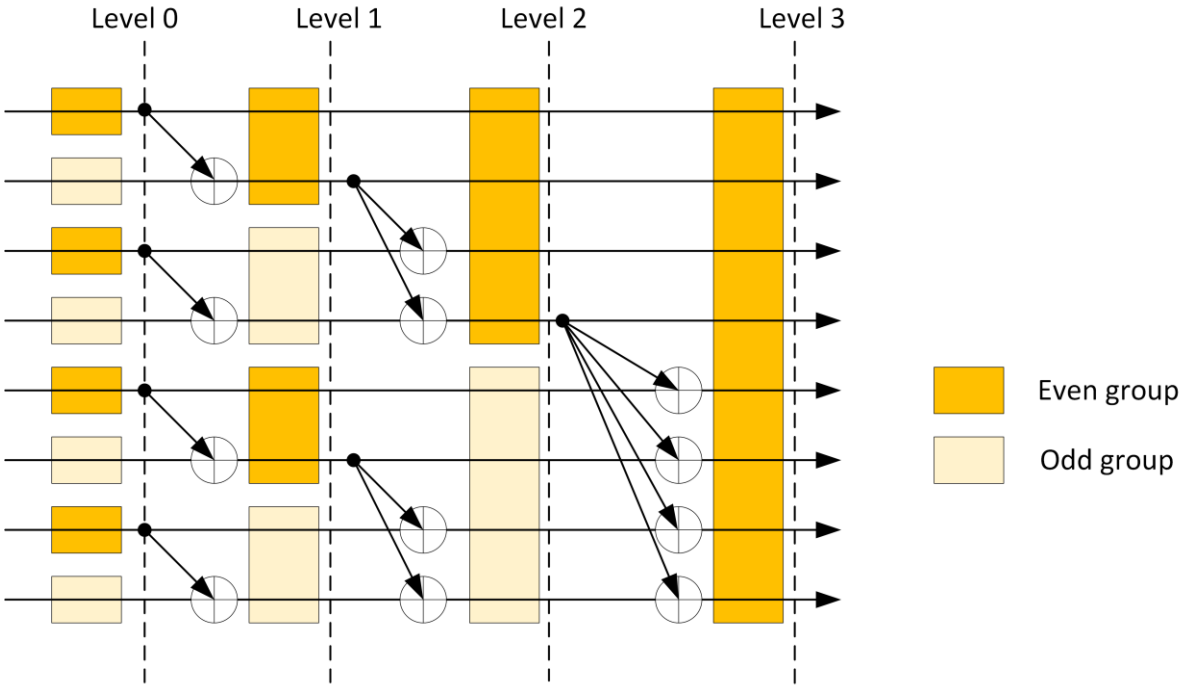

Figure 3.17: Parallel prefix computation OR tree

3.6.3 Extensions

For PCIe arbiter, which is the sticky arbiter, all that is need is to be done is ignore the right shifting of next mask vector calculated from current grant vector so that the current granted client has priority in the next cycle as well. For *area-optimized version*, the select logic is moved before the OR-PPC tree, this would only require a single arbiter, however additional delay of select logic is added. The arbiter can also be easily pipelined by grouping the levels into pipeline stages.

CHAPTER 4

EVALUATION AND RESULTS

This chapter outlines the testing procedure used, evaluation and interpretation of results in both hardware and software for unit tests of NVM Controller, PCI Express wrapper and then the integrated tests. It is to be noted, only preliminary testing for the blocks has been presented in this thesis, this does not qualify as a complete verification environment. In addition, ASIC synthesis results for NVM Controller are also presented at the end.

The FPGA used for the both the synthesis and emulation is *Xilinx ac701 evaluation kit* which consists of on-board PCIe Gen2 4x lane PHY and the PCI controller is a provided as a soft-core IP, 7 series integrated PCI Express. In all the tests below the 7 series integrated PCI Express is used with 128-bit data width and at a clock frequency of 125 MHz (125 MHz is because of 4 Gbit/sec speed per lane of PCI Gen2 post 8b/10b decoding).

4.1 TESTING OF PCI EXPRESS WRAPPER

The functionality of PCI Express wrapper is both tested in simulation and hardware. Figure 4.1 shows the simulation setup for unit test of PCI Express Wrapper. In simulation, towards the PCI core a testbench which delivers PCIe read packets and write packets to PCI Express interface is used and on the NVM Express side, a model with NVM Controller interfaces is created.

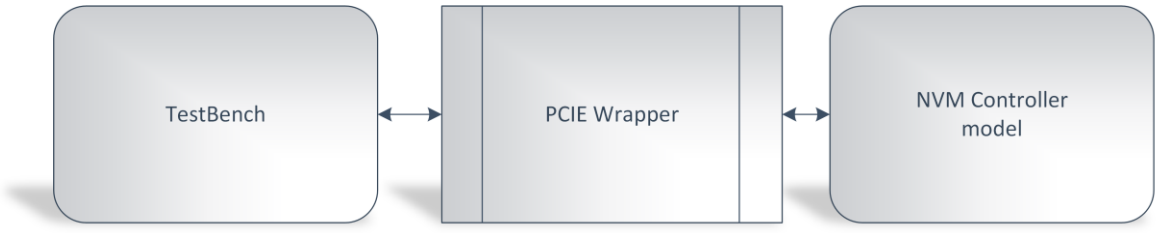

Figure 4.1: PCIe Wrapper simulation test environment

4.1.1 NVM Controller model

Though the model emulates the NVM Controller interfaces, the functioning of model is much simpler than that of the actual NVM Controller so that the bugs in NVM Controller doesn't affect that of the PCI Express wrapper. The model configuration interface consists of the following memory-mapped registers which control the data flow of the NVM controller model.

- **Address register:** It holds the pcie address for read/write data transfer.
- **Length register:** It holds the length of the data transfer.
- **Write register:** When this register is set, it implies that the data transfer is a write, if it is unset, it implies that the data transfer is a read.
- **Warps register:** This register is to be set only for read data transfer. For read data transfer, number of warps indicate the number of continuous data transfers that are initiated by the model even while the previous requests are outstanding. This register is used in the hardware to calculate request to response PCIe latency.
- **Start register:** When this register is set, the model initiates the data transfer based on the values set in all the previous defined registers. Hence this register should not be set until all the previous registers are configured properly.

4.1.2 Simulation flow

In simulation, the testbench configures the data flow control registers (*address, length, write* and *warps*) which are memory-mapped, by creating PCIe *write* TLPs and correct register setting is checked by creating PCIe *read* TLPs in order. Then the testbench initiates the data transfer by setting *start register*. The type of the data transfer can be configured by changing the values written to the data flow control registers by the testbench. In case the data transfer is a *read* (note that *read* here refers to read w.r.t model), the testbench responds with a completion packet consisting of random data and the data is checked for in the model BRAM where the read data is stored. In case of *write*, the data transfer is checked for the timing at the testbench interface.

4.1.3 Hardware emulation

In hardware, the configuration shown in the Figure 4.2 is used. The PCI Express wrapper is connected to Xilinx 7 series integrated PCI Express core towards the host interface and to the NVM Controller model towards the NVM Controller interface. On the host-side, a PCI Express driver which can be inserted in run-time in the linux kernel using kernel modules is developed. The following is the description of driver.

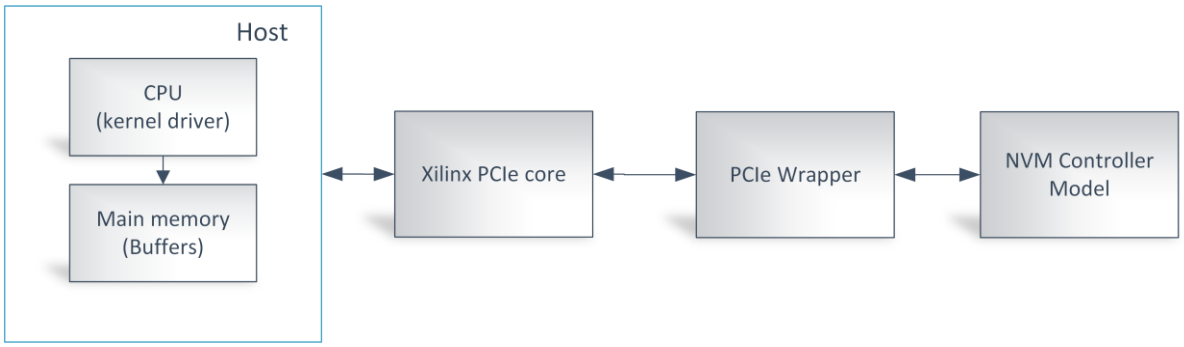

Figure 4.2: PCIe Wrapper hardware emulation environment

4.1.3.1 Kernel module testbench

The kernel driver makes use of linux PCI express API to talk to the FPGA emulated PCIe device. It initializes the data flow configuration registers which are memory mapped to PCIe BAR and checks that the configuration is set properly by reading them back using write and read API provided by linux kernel. Then the driver allocates DMA (Direct Memory Access) coherent write and read buffers in the kernel space through DMA APIs to/from which the data is transferred from/to the controller model via pci express wrapper. First a write data transfer is initialized and once the transfer is complete, a read data transfer is initialized, the read data is then compared with the written data for transmission errors, this validates that the preliminary PCIe wrapper functionality.

4.1.3.2 PCI Express speed test

Also PCI Express data rate and request-response time has been evaluated using the above described setup. For this purpose, additional hardware counter is instantiated in the NVM controller model which is also memory mapped to the configuration interface and hence can be read by the host. Before initiating the read data transfer, the counter is set to zero by the

kernel driver and the once the data transfer start register is set, the counter automatically starts counting and stops as and when complete data as per the length set in the *length* register is received by the model. This register is then read by the kernel driver and noted. The above process is repeated from 1 to 32 pages and a graph, shown in Figure 4.3, of number of clock cycles elapsed vs number of pages transferred is plotted. From this graph, the speed and latency can be calculated using the following formula.

$$Total\ cyles\ elapsed = cycles\ per\ page\ transfer * no.of\ pages + response\ latency$$

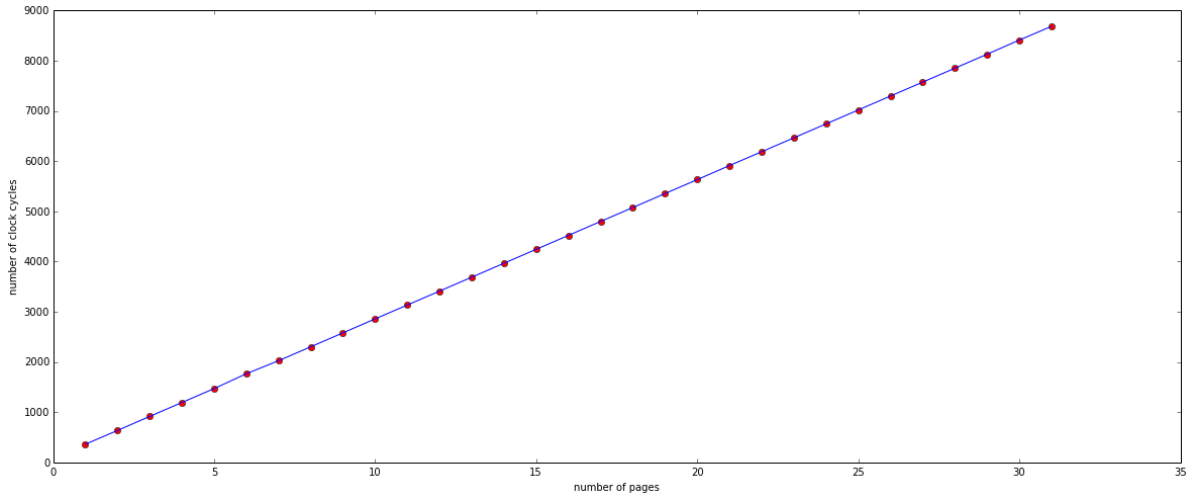

Figure 4.3: PCI Express speed test (number of clock cycles vs number of pages)

Hence the slope of the graph, cycles elapsed vs number of pages, gives the average cycles elapsed in 1 page data transfer and latency is given by the y-intercept. From the average cycles per page, the data rate can be calculated as

$$data\ rate = \left(4096 * 8 * \frac{clock\ frequency}{clock\ cycles\ per\ page(slope)}\right) Kbits/Sec$$

The *clock frequency* in this case is 125 MHz and the slope is obtained to be 277.4 cycles/page therefore the data rate is 14.76 Gbit/s and *latency* is obtained to be 83.04 cycles i.e., 0.664 uS.

Theoretical maximum speed of a PCIe Gen 2 4x is 16 Gbit/s, however in practice and in this particular setup on which the test is done, we obtain only 14.76 Gbit/s. Interestingly, the relationship between clock cycles and number of pages is very linear as seen in Figure 5.3 and fits our model perfectly !!. The reason is that the variations in PCIe latency is very few clock cycles and in the scale of our graph which is of 1000 cycles, the variation is quite small to observe. So, the latency looks almost constant.

4.2 TESTING OF NVM CONTROLLER

To test NVM Controller, we need to emulate the main memory and NAND flash. For this purpose a main memory model and NAND flash model is created using BRAMs in *Bluespec*. The testbench used is an extension of the testbench written in the previous work [17].

4.2.1 Simulation test environment

The simulation test environment is shown in the Figure 4.4.

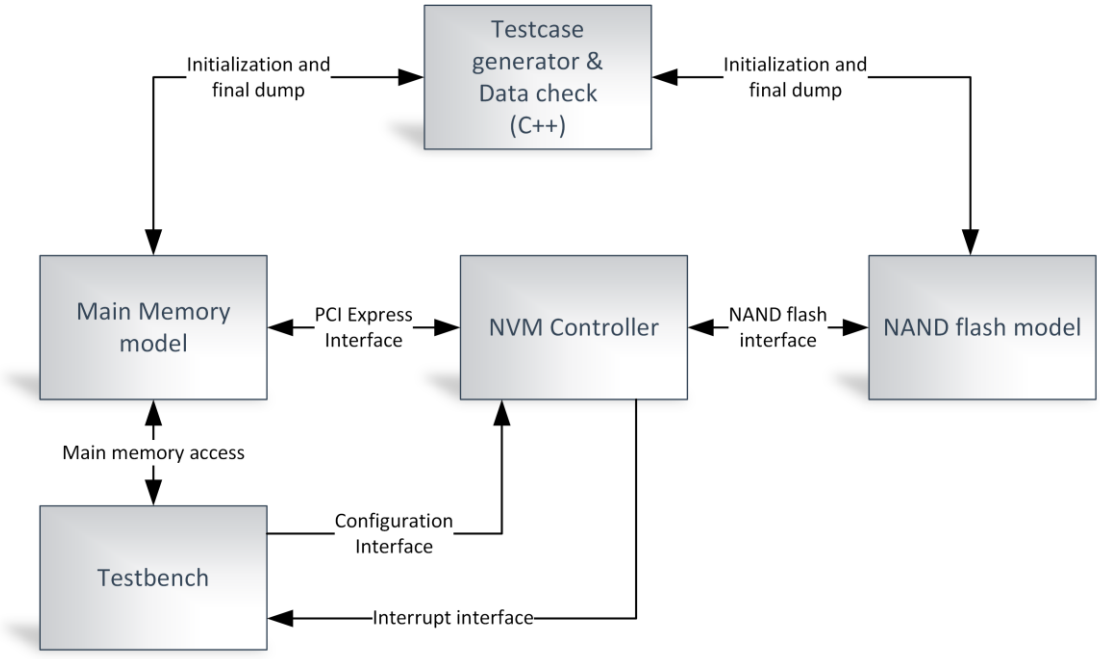

Figure 4.4: NVM Controller simulation environment

4.2.1.1 Main memory model

The *main memory model* consists of configurable sized dual-port BRAM which holds I/O submission queues and completion queues. One of ports is allowed for access to *NVM Controller* and the other port is used by the *testbench* for initialization, creation of I/O queues and checking completions. Since only one port is allowed for *NVM Controller*¸ unlike real environment both write and read from main memory cannot happen in parallel.

4.2.1.2 NAND flash model

The NAND flash model is also a configurable sized single port BRAM. It provides NAND flash interface abstraction to the *NVM Controller*. Every channel of *NVM Controller* is connected to one *NAND flash model* and all the channels of *NVM controller* can operate independently.

Apart from providing the storage space, it also provides an additional functionality to throttle NAND flash interface speed by enabling the rules and interface methods once in every *NO_CYCLE_DIVIDE* cycles. The parameter *NO_CYCLE_DIVIDE* is used to control the speed of NAND flash interface in order to emulate a real NAND flash which works at much lesser speed than the NVM Controller without creating the hassle of Clock Domain Crossing. This speed throttling is used in evaluating the performance of multi-channel controller and hence giving us an insight into the controller limitations etc.

4.2.1.3 Test case generator

The test case generator is written in *C++* which creates the initialization data in the *main memory* including random I/O commands, random addresses and random data based on the available main memory size and NAND flash size. The number of commands, number of pages per command and type of commands are parametrized in the test case generator and hence it can generate test cases for different scenarios. The I/O queues are given a fixed address map and the test case generator places the I/O commands in the respective addresses of the main memory. It also creates random data for NAND flash initialization. The generated main memory and NAND flash contents are exported to a text file which is later used by the *top level testbench.*

4.2.1.4 Top level testbench

The top level testbench instantiates the main memory and NAND flash model, and connects them appropriately, the number of channels and bit width of the testbench is also parameterized. The instantiated main memory model and NAND flash model are initialized with the contents generated by the *test case generator.* The top level does the initialization of the NVM Controller like setting admin submission queue and completion queue base addresses, setting controller capabilities and checking the set capabilities etc. Once the initialization is done, it creates the I/O submission and I/O completion queue based on the address map used by the *test case generator* and writes the submission queue tail doorbell to the controller configuration registers. Since the I/O commands are already initialized in the I/O queues present in the main memory, the controllers starts fetching the commands and processes it. The completions sent by the controller are also checked for their status by the top level testbench.

4.2.1.5 Post-processing data check

The post-processing data check is also a script written in *C++*. It takes in the initial main memory and nand flash contents generated by the *test case generator,* final main memory and nand flash contents generated by the *top level testbench*, executes the I/O commands in main memory and compares the final data with the expected data.

4.2.1.6 Limitations and extensions

This kind of approach for verification has known limitations and hence needs to be extended. Firstly, the I/O commands are not en-queued in the I/O queues at run time but are initialized in the main memory and are never altered. So, the maximum number of I/O commands is limited by the I/O queue size whereas in real situations the executed commands are replaced with new commands through wrap around functionality of the queues. This limitation however can be alleviated by adding extra logic to enqueue the commands generated by the *test case generator* in run time, first by initializing a command buffer with the generated commands and enqueuing the commands into main memory queues at run time.

Secondly, the data check is also not carried out at run-time, therefore this requires that the entire execution to be completed even if the error resides much before the completion, this consumes lot of simulation cycles. Also, the final main memory contents needs to dumped

into a text file for comparison, however, unlike Verilog or VHDL languages, Bluespec doesn't have any methodology to dump entire BRAM contents at one shot, we will need to traverse through the entire BRAM one by one and print the contents to a file. This again consumes lot of simulation cycles. This however is not a limitation of the approach but is the limitation of Bluespec !!.

4.2.2 Simulation speed tests

Using the environment described in the earlier section, the NVM controller is evaluated for its performance. *Read* and *write* commands with various page sizes and various commands for each size is generated using the *test case generator* and the simulation cycles is measured from the write of door-bell register to the completion of all the commands. In this simulation, the NVM controller is assumed to be the bottleneck and hence NAND flash runs at same speed as the whole design, while in practice the throughput is almost certainly limited by NAND flash bandwidth. However, this simulation gives an insight into the overhead incurred by the design and the NVM Express protocol. Figure 4.5 shows the maximum speed that can be obtained using the controller for *read* and *write* commands of different page sizes.

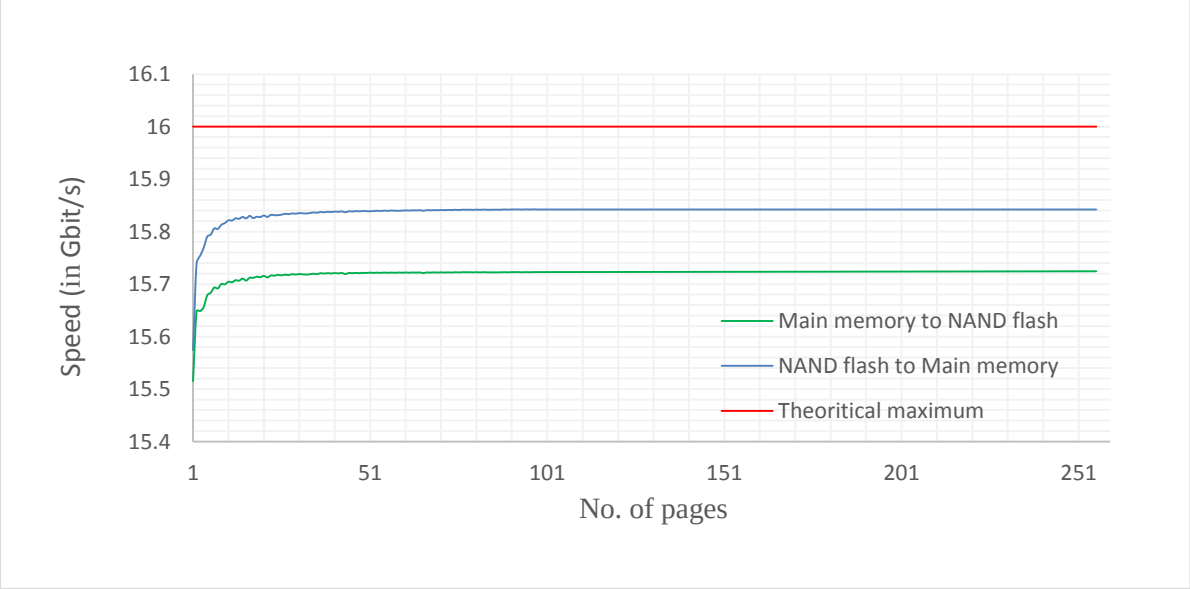

Figure 4.5: NVM Controller simulation speed test

Initially, for smaller page I/O, the speed is lesser because the percentage of overhead introduced due to fetch, execute and PRP fetch stages when compared to the data transfer is quite higher. For any command, the overhead encountered due to fetch and execute is constant, however the overhead encountered due to PRP fetch increases proportionally. When compared to the fetch and execute overhead, the PRP fetch overhead is significantly lesser. As the number of pages increases, the percentage of overhead incurred due to the significant constant part decreases and hence the speed increases. Also, we need to observe that the variation from small I/O to large I/O is very less and quickly increases to maximum value. Also, the *write* speed is lesser than the *read* speed, this is because of an extra response to request cycle from main memory simulation environment and has nothing to do with the controller itself. However, request to response latency is typical of any I/O interface and is much larger than a single cycle in practice, this is hidden by *channel data buffer* of NVM controller. In this case, the latency is not hidden because the NAND flash also operates at same speed as the NVM controller, while in practice, NAND flash interface is the bottle neck and the extra latency will be hidden.

The actual speed of the controller depends on the speed of the I/O interface. In this case, the I/O interface is taken to be PCIe Gen 2 4x interface with maximum theoretical speed of 16 Gbit/sec and we get a maximum speed of 15.83 Gbit/s. A better metric that is independent of the speed of the I/O interface, is the percentage of bandwidth utilization which is given by

$$\%Bandwidth\ utilization = \frac{Controller\ bandwidth}{IO\ inteface\ bandwidth}$$

In this case, the maximum percentage bandwidth utilization is (15.83/16) ~ 98.9%. This when combined with the hardware PCI express Gen 2x practical evaluation in the previous section gives a maximum speed of 14.6 Gbit/s. This however, is not real hardware evaluation and only an estimate. The real hardware speed evaluation is not possible with current test driver because the current test driver cannot handle multiple outstanding requests and is to be evaluated with complete *LightNVM* stack and forms part of the future work.

In order to evaluate, the performance of the controller in the real situation in which NAND flash interface is the bottleneck, the functionality of the NAND flash model to throttle the

speed of NAND flash interface as specified in section 4.2.1.2 is used. The NAND flash interface is throttled by configuring NO_CYCLE_DIVIDE to 25 and since the simulation is run at 125 MHz, cycle divide of 25 implies that the NAND flash is operating at 5 MHz. Along with 128-bit data-width, it emulates a NAND flash interface working at an average of 5*128 Mbit/s = 640 Mbit/s. Figure 4.6 shows the normalized NAND flash interface utilization with two values of *channel command buffer size.* The utilization is normalized to single channel and for easy comparison. The interface utilization for single channel controller is obtained to be 99.9%.

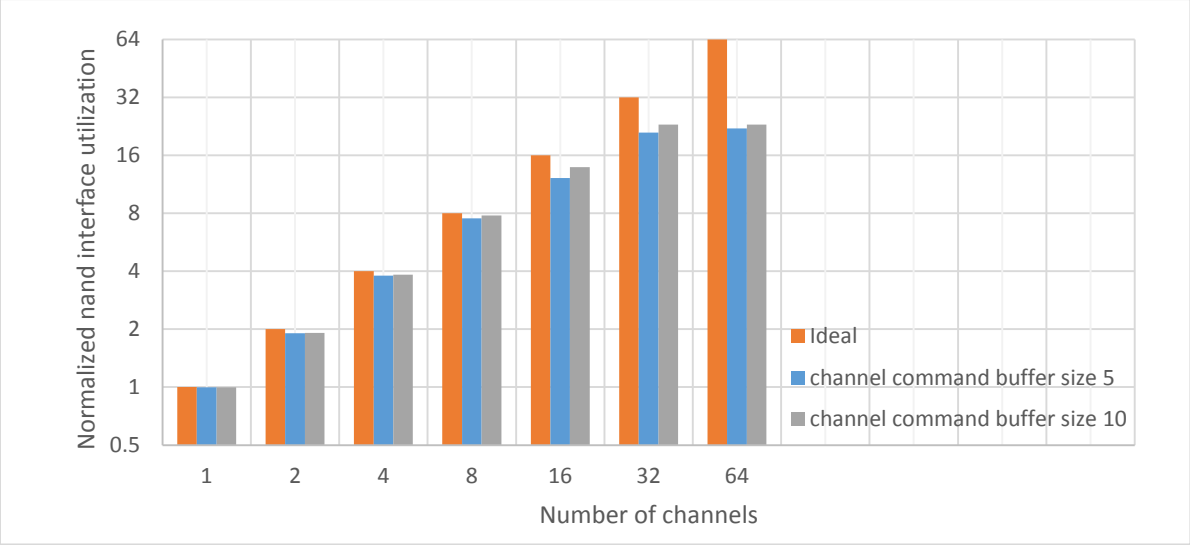

Figure 4.6: Normalized NAND interface utilization vs number of channels

Ideally, if the number of channels doubles, the NAND interface utilization should also double. As we can see from Figure 4.6, the actual utilization deviates from the ideal as the number of channels increases for two reasons.

For the case in which number of channels is less than 16, the deviation is because of channel buffer skewing. That is the test cases are random and hence are globally uniformly distributed across the channels. However, locally the I/O commands need not necessary be uniform and this results in channel skewing i.e., when I/O commands are locally skewed towards one or more channels, their channel buffers becomes full, halting the command processing in other channels. This results in reduced average interface utilization and this

effect is pronounced more as the number of channels increases because of the increase in number of halted channels. Increasing the size of *channel command buffer* reduces skewing and hence the channel utilization increases.

For number of channels greater than 16, since the NAND flash interface is emulated at 640 Mbit/s which is 1/25th of NVM controller bandwidth which is 16 Gbit/s, beyond 16 channels, the NVM controller becomes the bottleneck and hence the utilization doesn't increase.

4.2.3 Hardware emulation

4.2.3.1 FPGA test configurations

In hardware, two configurations are used for testing. The first configuration is a straight data path from PCIe to NVM controller to NAND flash model as shown in Figure 4.7. This configuration is similar to the simulation environment except that the main memory model is replaced with actual main memory and the NAND flash model is still constructed with BRAMs. The number of BRAMs in the NAND flash model is however limited because of resource constraints in an FPGA. So this configuration can only be used with a maximum of 4 pages.

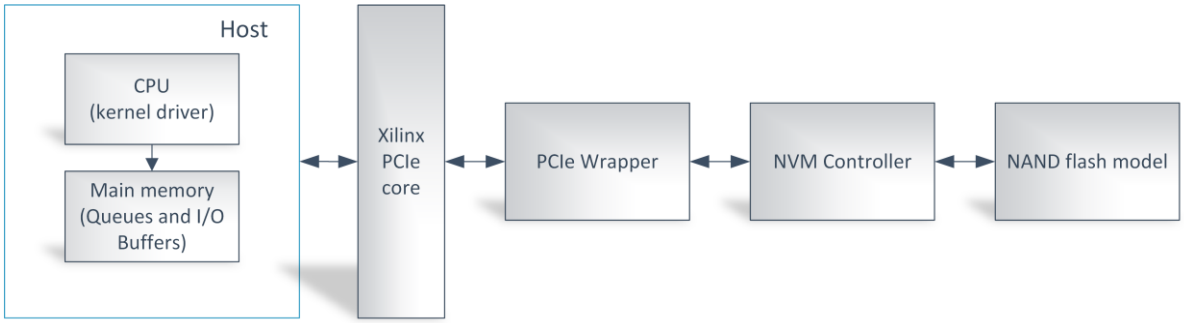

Figure 4.7: NVM controller hardware test configuration 1

In the second configuration shown in Figure 4.8, the NAND flash model is emulated with an on-board DDR3. This is done by creating an AXI4 stream wrapper around the NVM Controller NAND flash interface and is connected to Xilinx AXI datamover which converts AXI4 stream to AXI4 lite without any additional latency. The AXI4 lite from *Xilinx AXI datamover* is connected to an AXI interconnect including other peripherals like Microblaze,

AXI UARTlite, DDR3 etc The number of pages in this approach is only limited to the DRAM size which is around 1 GB and is sufficient for our purposes. Also, it provides more visibility into the hardware than the earlier one because of on-board microblaze.

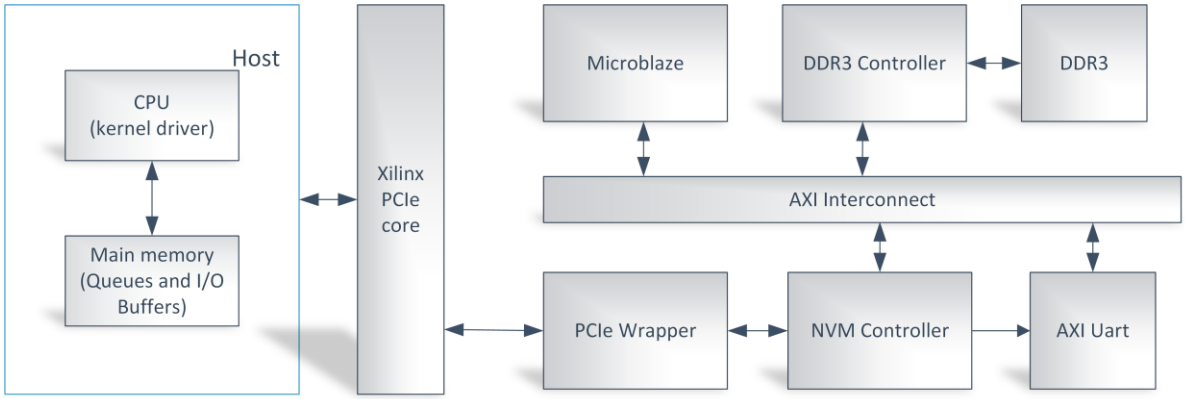

Figure 4.8: NVM controller test configuration 2

In both the configurations, similar to the kernel testbench in FPGA emulation of PCIe wrapper and NVM controller model in section 4.1.3.1, a linux device driver is created that adheres to NVM Express specifications. The following section describes the kernel module and testing process.

4.2.3.2 Linux kernel module

The kernel module is built with the linux nvme driver as a reference but removing the interface to the user space so that the data flow is in the control of the driver. Similar to the functioning of the *top level testbench* in simulation, on insertion of the kernel module using "*insmod",* the driver attaches itself to the underlying device based on the device Id and vendor Id of the pcie core. The driver then allocates DMA coherent memory for admin I/O queues in the kernel space and writes the queues base addresses to the memory mapped registers of NVM controller configuration interface as specified by the NVM Express specification. Then the driver creates the I/O queues and DMA coherent receive and send buffers and submits *create completion queue* and *create submission queue* commands to the admin queue and waits for the completion. The completion is checked using phase tags (see section 2.4.1) since interrupts are not yet implemented in the PCIe wrapper. This is for the reason that an unexpected interrupt can simply hang the kernel without any information for the cause of error and hence

it is decided that the completion be checked with phase tag functionality initially and then implement interrupts.

Once the I/O queues are created, the driver initializes the send buffer with random data and submits a write command to the I/O queue and on completion the same data is read back into the receive buffer and compared with the send buffer for correctness. In test configuration 2, the microblaze is also used to monitor the data received on the hardware side and this information is sent to host through AXI UART lite.

4.3 Synthesis results

The NVM Controller including the PCI Express wrapper is evaluated for maximum clock frequency and area in ASIC synthesis. All the results below are obtained using *Cadence RTL compiler* and *UMCIP 65 nm library* at operation conditions of 1.35V and 110^0F.

Figure 4.9 shows the maximum clock frequency of the NVM Controller with PCI Express wrapper when configured with various channels and I/O queues.

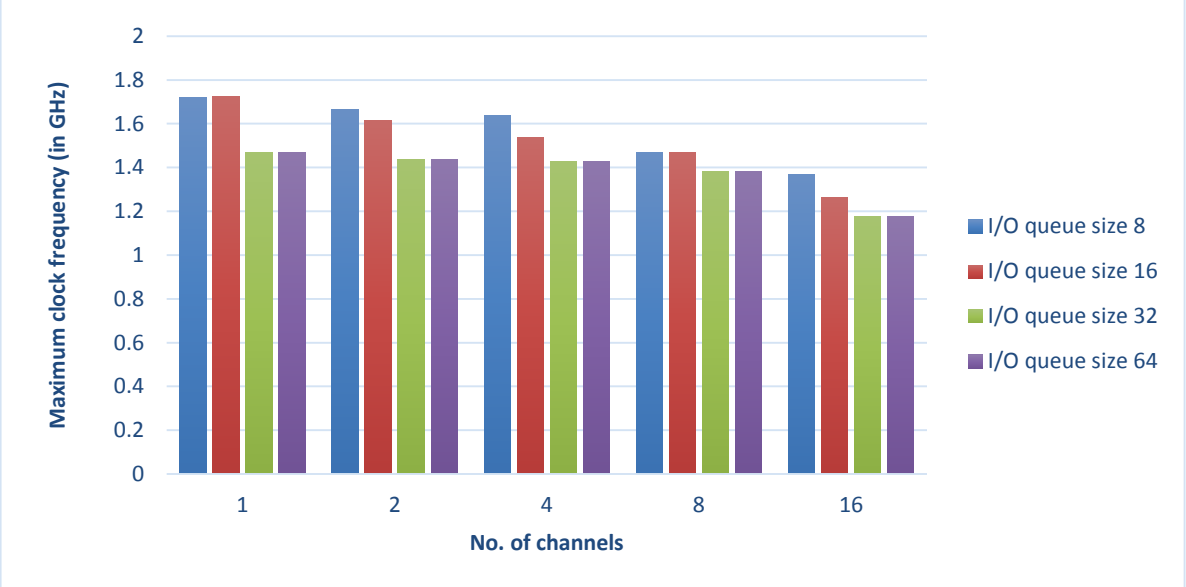

Figure 4.9: NVM controller maximum clock frequency Vs No. of channels and I/O queues

The maximum clock frequency decreases with increase in number of channels and number of I/O queues as expected. However, note that given an I/O queue size, the decrease is clock

frequency with no. of channels is not very significant when compared to the performance increase due to increase in number of channels. For eg. for I/O queue size of 16, the clock frequency decreases from 1.72 GHz to 1.26 GHz i.e., 0.73x decrease, whereas the performance increases by approximately 16x from single channel to 16 channels. Overall, the clock frequency is above 1 GHz for all the above cases and with 128 bit data width and 98.9% bandwidth utilization (see section 4.2.2), the controller can deliver speeds as high as 100Gbit/s. Also, note that the clock frequency, given a fixed number of channels, almost remains same for 8 and 16 I/O queues and decreases at 32 I/O queue size and again remains the same for 32 and 64 I/O queues. This is because, for queue size greater than or equal to 32, the I/O queue arbiter forms the critical path and hence the clock frequency is determined by the delay of the arbiter logic and this is why the design of arbiter logic is significant in determining the performance.

Figure 4.10 shows the clock frequency of NVM controller using round robin arbiter provided in *Bluespec library* and the one designed in section 3.6 of this thesis, for I/O queue size of 32, 64 and 128 at which the arbiter logic is the critical path with a fixed number of 8 channels per controller.

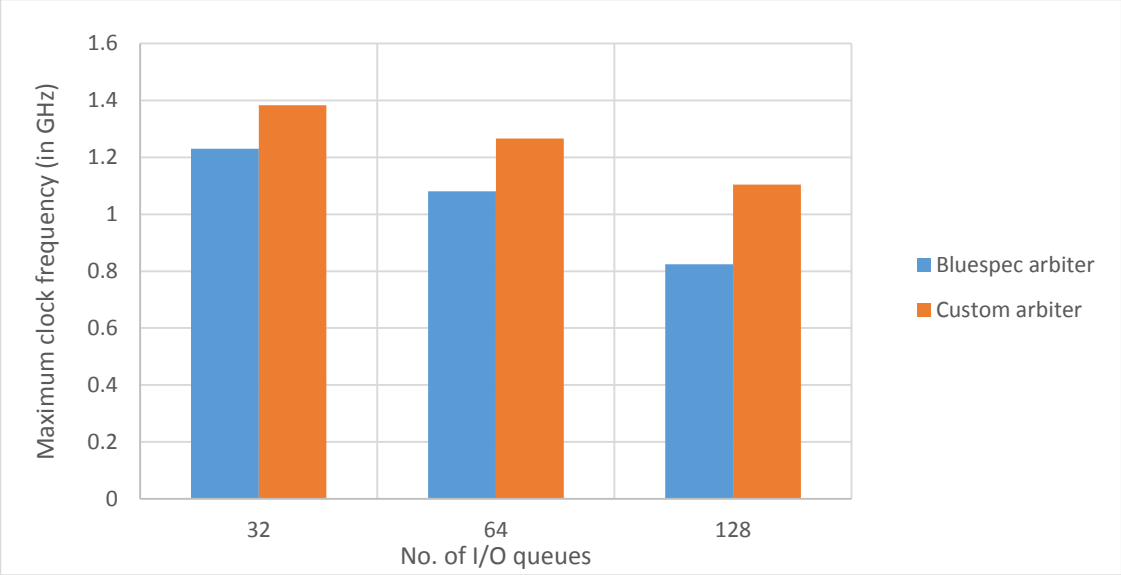

Figure 4.10: Performance of BSV arbiter vs Custom arbiter

As we can see, the maximum clock frequency achieved by using BSV arbiter is less than that of the custom arbiter in all the three cases. But more importantly, the decrease in the frequency with BSV arbiter as the number of I/O queues increases is much more than that of the custom arbiter. From queue size of 32 to 128, the BSV arbiter decreases from 1.21 GHz to 0.82 GHz which is 47.5% reduction whereas the custom arbiter reduces from 1.38 GHz to 1.10 GHz which is only 25% reduction. The BSV arbiter doesn't scale well with increase of I/O queues because of *O(N)* delay whereas the custom arbiter has *O(logN)* delay.

Figure 4.11 shows the variation of combinational area with channels, for a fixed I/O queue size of 64 and clock frequency of 1GHz and Figure 4.12 shows the variation of sequential area with channels.

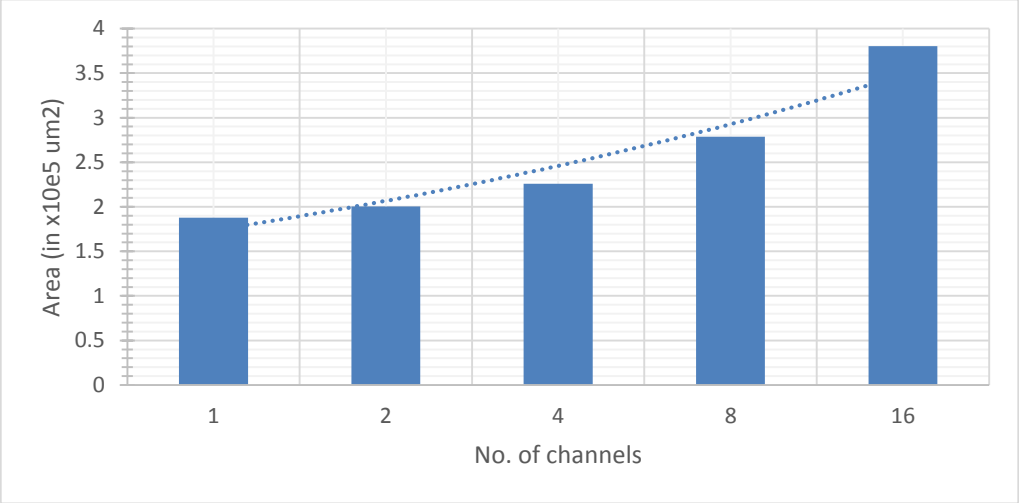

Figure 4.11: Combinational area vs No. of channels

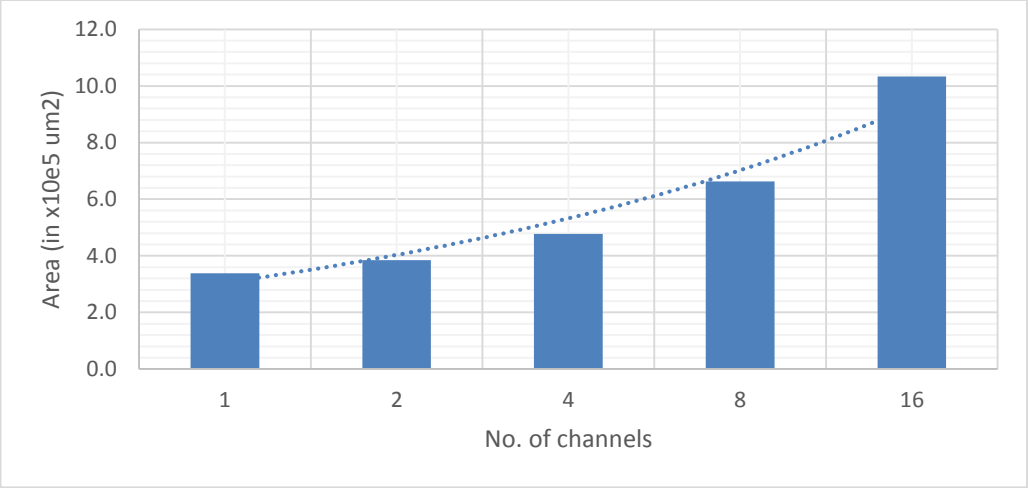

Figure 4.12: Sequential area vs No. of channels

As expected, the sequential and combinational area increases with increase in number of channels. In this case, the absolute value of sequential area is not of much importance since the buffers are currently synthesized using Flip-flops, while in practice, they would need to be synthesized using SRAM cells. However, the point of importance is the trend of area increase. The *trendline* in Figure 4.11 and Figure 4.12 shows ideal proportional increase of area w.r.t channels and as we can see the actual area closely matches this. Now, this in combination with the results of clock frequency proves that the designed controller is indeed scalable.

CHAPTER 5

EXTENSIBILITY AND FUTURE WORK

This chapter begins by outlining some of the general design choices that have been taken in designing generic NVM controller and possible extensions. A summary of what this thesis has achieved is presented towards the end followed by future work.

5.1 GENERAL DESIGN CHOICES AND EXTENSIONS

5.1.1 Global data buffer vs Distributed data buffer

Traditional implementation of storage controller use data buffer for two reasons. Firstly, to hide the host I/O interface latency and then it can be also used as a volatile cache for frequently accessed pages. Current implementation has one data buffer per channel (*distributed data buffer)* and the size of which is to be determined by the host I/O interface latency and NAND flash channel speed, however, a single *global buffer* for all the channels combined is also a possible implementation.

A single global buffer has some disadvantages in terms of performance. Firstly, since the global buffer is shared across channels, skewed I/O requests can severely affect the performance of other channels. This is because, if I/O requests are skewed towards some of the channels, the global data buffer gets quickly filled and the rest of the channels will remain idle until there is enough space in the data buffer. However, a distributed data buffer is resilient to this because if there are skewed I/O requests, once the corresponding channel data buffers gets full, simply the data fetch for those particular channels will halt and rest can proceed without disturbance.

Secondly, in case of global buffer, since the address space is shared across channels, additional arbitration logic is to be incorporated across the channels post-data fetch and since throughput of the data buffer should match the channel throughput, the data buffer should

operate at higher frequency than the controller itself necessitating clock domain crossing and synchronization logic. This additional overhead is avoided in the distributed data buffer.

However, global buffers can be advantageous when FPGA based storage controller is designed since most of the commercially available FPGAs have on-board DRAMs which can act as shared data buffer. The distributed data buffer on FPGAs would need to be implemented using BRAMs but this is limited by the amount of BRAM size of FPGA. Another way to implement distributed data buffer is to allocate fixed address spaces to each channel but then the advantage of not requiring arbitration and clock synchronization logic will be lost. The second implementation is recommended for FPGAs, if the data buffers cannot be implemented using on-board BRAMs and is directly supported by current controller.

5.1.2 External interface protocol

The external interface plays an important role in extensibility and flexibility of operation with the external peripherals. Currently, *Bluespec* based interface is provided towards the NFC. This enables the NFC written in *Bluespec* to easily interface with the NVM Controller. Also, the interface is so designed so that no additional complexity needs to be added to translate the interface to standard interface like AXI for interaction with standalone IPs. AXI wrappers around the NFC are also written and used in the testing of NVM controller without any additional latency and can be extended for interfacing with standalone IPs.

5.1.3 Impact of buffer sizes on performance

NVM controller offers several design parameters that can be configured especially the buffer sizes. Overdesign of buffer sizes would simply waste the chip area and increase power consumption however under-design would impact performance. Hence the impact of buffer sizes is briefly provided here for later calibration.

The size of *Channel command buffer* directly impacts the resilience of the controller to skewed I/O requests. The multi-channel FTL can ensure that the commands are globally uniformly distributed across the channels whereas the resilience towards local distribution is provided by the size of this buffer. If the size of this buffer is too small, a local skewing of I/O commands towards a subset of channels could halt the execution of commands by other channels, if the size of this buffer is too large, it would simply waste the chip area.

The size of *Command completion ROB* should at least be equal to the number of outstanding commands in the complete pipeline. If it less than that, it prevents the buffers in the pipeline from being utilized completely. The current *command completion ROB* uses a circular buffer implementation. In this implementation, the size of the buffer should be large enough to tolerate the maximum delay of command completion by a channel NFC else it would block the execution of commands by other channels because of the buffer being full. In case the implementation is extended to using *free register queue* and allow out of place completion delivery to host, then the buffer size should be same as number of maximum outstanding commands in pipeline.

The size of *channel data buffers* should be enough to hide the data fetch latency of the host I/O interface.

5.1.4 Extending for out-of-order command execution

The current implementation provides for out of order execution of commands targeted across channels in order to not block any channel from execution because of any other channel. This is implicit in the current pipeline architecture however the commands targeted at a single channel are executed in order. It could be advantageous to provide for out-of-order command execution within the channel for better performance (for eg: certain sequential reads given more priority than writes etc.). The generic controller can be extended for this purpose.

Firstly, the *command completion ROB* must be implemented using *free register queues* instead of *circular buffer*. The *free register queue* would maintain the free entries in the command ROB and the completed entries can be ejected out of order as when the I/O commands has been completed. Additional select logic similar to select logic presented in *arbitration tree* should also be added for ejection. All the I/O commands that are ready to be ejected will request the *arbitration tree* and one of them gets the grant based on the position and once the entry is ejected, the entry id is added to *free register queue.* This implementation would add extra complexity and hence is left optional depending on the use case.

Then, the PRP buffers are to be moved into the *command completion ROB*, currently they are provided by FTL processor, with fetch-on-demand logic implementation and expose the command tag to underlying NFC.

5.2 SUMMARY AND CONCLUSIONS

In this thesis, the concept of *generic multi-channel storage controller* is explored, architected and implemented. The entire design is focused in promoting an extensible and interoperable platform. Also generic interfaces supporting host-side FTL and device-side FTL have been added. Various design parameters are exposed for use case specific tuning. An elaborate test environment with automatic random test case generators and simulation models written in *Bluespec* and *C++* have been developed. In addition, PCI Express wrappers and linux kernel modules have been developed for testing of the controller in an FPGA environment. The designed controller is evaluated for its performance and limitations.

It has been shown that the current controller utilizes 98.9 % of the host I/O interface bandwidth, can potentially provide bandwidth as high as 100Gbps with 65nm ASIC design and is highly scalable. The designed controller outperforms all commercially available storage controllers, in terms of functionality and genericity, up to date and hence forms the high performance storage controller part for much bigger vision of *LightStor* project with a complete SOC consisting of on-board processor, encryption and deduplication engines.

5.3 FUTURE WORK

The current system forms a much simpler version of the controller than that envisioned by the *Lightstor* project. In the current thesis, the NAND flash controller and NAND flash are abstracted out and emulated using BRAMs and DDR. However, the controller needs to be integrated with NAND flash controller and tested with actual NAND flash model. The controller only contains preliminary support for *LightNVM* specification. The support for hybrid I/O and storage of translation tables in the controller through a cache hierarchy is to be added. In the current implementation, the processor is used only for debugging purposes, however the processor can be integrated with the NVM controller command flow to support additional commands like SMART, LOG etc or extended support for databases. The current controller is tested with a preliminary kernel driver supporting simple data transfers, however, it needs to be tested with actual *LightNVM* software stack, including host-side FTL layer.

Other extensions could be replacing the PCI Express host interface with RapidIO interface, adding on-board peripherals like encryption and deduplication engine.

BIBILIOGRAPHY

[1] Wikipedia. Floating gate MOSFET, 2015.URL:http://en.wikipedia.org/wiki/Floating-gate_MOSFET

[2] Wikipedia. Flash memory, 2015. URL: http://en.wikipedia.org/wiki/Flash_memory

[3] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., & Cho, Y. (2002). A space-efficient flash translation layer for CompactFlash systems. *Consumer Electronics, IEEE Transactions on*, *48*(2), 366-375.

[4] Lee, S. W., Park, D. J., Chung, T. S., Lee, D. H., Park, S., & Song, H. J. (2007). A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, *6*(3), 18.

[5] Gupta, A., Kim, Y., & Urgaonkar, B. (2009). *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings* (Vol. 44, No. 3, pp. 229-240). ACM.

[6] Kwon, O., Koh, K., Lee, J., & Bahn, H. (2011). FeGC: An efficient garbage collection scheme for flash memory based storage systems. *Journal of Systems and Software*, *84*(9), 1507-1523.

[7] Meng, W., Kai, B., Qiyou, X., Zhaolin, S., Xin, X., & Hui, X. (2012, January). An efficient NAND flash garbage collection algorithm based on area and block operation. In *Intelligent System Design and Engineering Application (ISDEA), 2012 Second International Conference on* (pp. 1192-1195). IEEE.

[8] Chang, L. P. (2007, March). On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing* (pp. 1126-1130). ACM.

[9] Chang, Y. H., & Chang, L. P. (2013). Efficient Wear Leveling in NAND Flash Memory. In *Inside Solid State Drives (SSDs)* (pp. 233-257). Springer Netherlands.

[10] Gao, C., Shi, L., Zhao, M., Xue, C. J., Wu, K., & Sha, E. H. M. (2014, June). Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (pp. 1-11). IEEE.

[11] Matias Bjorling, Javier Gonzalez, Jesper Madsen Open channel SSD, 2015 , URL: http://www.linuxplumbersconf.org/2014/ocw/sessions/2079

[12] NVM Express Inc. NVM Express spec, 2015 URL: http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf

[13] PCI-SIG, PCI Express Specification Base specification 3.0, 2015. URL: https://www.pcisig.com/specifications/pciexpress/

[14] Wikipedia. PCI Express, 2015. URL: http://en.wikipedia.org/wiki/PCI_Express

[15] ONFI specification, 2015. URL: www.onfi.org/specifications

[16] Wikipedia. RapidIO protocol. URL: http://en.wikipedia.org/wiki/RapidIO

[17] Maximilian Singh, Master thesis, Development of a FPGA Based Programmable Storage Controller, 2014

[18] Xilinx Inc. 7 series integrated PCI Express core version 3.1 (pg054) .URL: http://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_1/pg054-7series-pcie.pdf

[19] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers 3rd edition. URL: https://lwn.net/Kernel/LDD3/

[20] G S Madhusudan, Matias Bjø rling, Jesper Madsen, and Philippe Bonnet. The Lighstor Storage System, 2014. URL: http://www.lightstor.org/

[21] G S Madhusudan, Matias Bjø rling, Jesper Madsen, and Philippe Bonnet, 2014. URL: https://github.com/lightstor/specification/blob/master/lightstor_overview.pdf

[22] Bluespec Inc. Bluespec System Verilog, 2013. URL: http://www.bluespec.com/high-level-synthesis-tools.html#bsv-lang.

[23] L. C. Rino Micheloni and A. Marelli, Inside NAND Flash Memories. Springer, 2010

[24] Kang, J. U., Kim, J. S., Park, C., Park, H., & Lee, J. (2007). A multi-channel architecture for high-performance NAND flash-based storage system. *Journal of Systems Architecture*, *53*(9), 644-658.

.