UiT THE ARCTIC UNIVERSITY OF NORWAY

The Faculty of Science and Technology
Department of Computer Science

Improving Disk Performance in Vortex With NVMe

—

Kristian Elsebø

INF 3981 — Master's Thesis in Computer Science — June 2015

“Well, that was pointless. . . ”
–Hubert Farnsworth, Futurama

Abstract

With the development of SSDs, performance limitations in persistent storage have shifted from the underlying medium to the interface through which the host and disk communicates. NVMe is a recently developed standard for operating SSDs connected to a host through PCI Express, and offers significant performance improvements compared to conventional interfaces, as well as features designed for multi-tenant environments.

Vortex is an experimental implementation of the omni-kernel architecture, a novel operating system kernel designed to offer strong isolation and accurate, fine-grained scheduling of system resources for all tenants that share a platform. The BIOS of the hardware platform currently supported by Vortex does not recognize NVMe devices, and the Vortex operating system does not support configuration of devices that are unrecognized by the BIOS. Further, the storage stack implemented in Vortex only supports SCSI-based storage devices.

This thesis presents the implementation of an NVMe driver for Vortex that is exposed as a SCSI device. We also implement a system for recovering information about devices that are unrecognized by the BIOS, and use this system to successfully configure NVMe devices on our hardware platform. The NVMe driver is fully functional, deployed in a running Vortex system, and evaluated through performance experiments.

Acknowledgements

I want to thank my supervisor, Dr. Steffen V. Valvåg. Your advice and guidance through this project has been invaluable.

Thank you Dr. Åge “The Manual” Kvalnes for taking time off to share your knowledge.

A special thank you to Erlend Graff, your devotion and eagerness is highly contagious! Further, I want to express my sincerest gratitude to my fellow students and friends, Vegard Sandengen, Bjørn Fjukstad, Einar Holsbø, Jan-Ove “Kuken” Karlberg, Michael Kampffmeyer, and Magnus Stenhaug. To all of the above, I value your friendship and all the help (criticism) you have given me.

Also, a great big thanks is in its place to the backbone of IFI, Jan “The Man” Fuglesteg, Maria W. Hauglann, and the rest of the TK and administration stab. You all make this a great place to be a student!

And of course, to my parents, Kari and Nils-Jacob, and my sister Elisabeth: thank you for your support, I could not have done this without you.

Last, but not least, to my girlfriend Marita Skogmo: thank you for your love and for believing in me, and especially for sticking with me through the insanity. You have created an environment in which I am able to relax and recover during stressful times.

Contents

List of Figures

List of Code Snippets

List of Abbreviations

AER	Advanced Error Reporting
AHCI	Advanced Host Controller Interface
ANSI	Americal National Standards Institute
API	application programming interface
APIC	Advanced Programmable Interrupt Controller
AQ	admin queue
ASQ	admin submission queue
ATA	AT Attachment
ATAPI	AT Attachment Packet Interface
BAR	base address register
BDF	bus device function
BIOS	Basic Input Output System
CD	Compact Disk
CPU	central processing unit
CQ	completion queue
DBMS	database management system

DMA	direct memory access
DRAM	dynamic RAM
ECC	error-correcting code
EMI	electromagnetic interference
ESDI	Enhanced Small Device Interface
FS	file system
FTL	Flash Translation Layer
Gbit/s	gigabit per second
GPU	graphics processing unit
GT/s	gigatransfers per second
HBA	host bus adapter
HDD	hard disk drive
I/O	input/output
IDE	Integrated Drive Electronics
IOPS	I/O operations per second
IOQ	I/O queue
IOSQ	I/O submission queue
IRQ	interrupt request
ISA	Industry Standard Architecture

LBA	logical block address
LUN	logical unit
MFI	MegaRAID Firmware Interface
MP	MultiProcessor
MSI	message signaled interrupts
MSI-X	MSI extended
NAND	not and
NIC	network interface card
NVM	non-volatile memory
NVMe	Non-Volatile Memory Express
OKA	omni-kernel architecture
OS	operating system
P-ATA	Parallel ATA
PBA	physical block address
PC/AT	Personal Computer AT
PCB	printed circuit board
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PCI-X	Peripheral Component Interconnect eXtended
PIO	programmed I/O

PM	power management
PnP	Plug and Play
PQI	PCIe Queueing Interface
PRP	Physical Region Page
RAID	redundant array of independent disks
RAM	random access memory
RPM	rounds per minute
RR	round robin
SAS	Serial Attached SCSI
SASI	Shugart Associates Systems Interface
SATA	Serial ATA
SATA Express	Serial ATA Express
SCSI	Small Computer System Interface
SLA	service level agreement
SOP	SCSI over PCIe
SQ	submission queue
SRAM	static RAM
SSD	solid state drive
Tb/in^2	terabits per square inch

USB	Universal Serial Bus
VM	virtual machine
VMM	virtual machine monitor
WRR	weighted round robin

1 Introduction

In 1965, Gordon E. Moore observed that the number of components per integrated circuit had increased at an exponential rate, roughly doubling each year [1]. He conjectured that this trend would likely continue for at least ten years. For central processing units (CPUs), random access memory (RAM), and CPU-cache[1] technology, we see that this conjecture, known as "Moore's law", is still highly applicable.

The trend is particularly clear in the development of non-volatile memory (NVM). Over the past 20 years, the storage capacity for a single device has increased from a maximum of 1000 MB [2], to passing the 10 TB mark [3]. For conventional hard disk drives (HDDs), Moore's law is reflected in the areal density on the magnetic surface of a HDD platter, which currently peaks at 0.848 terabits per square inch (Tb/in^2) [4]. According to a roadmap released by the Advanced Storage Technology Consortium at the Magnetism and Magnetic Materials conference in 2014, the density is expected to reach 10 Tb/in^2 in 2025, providing approximately 100 TB of storage on a single HDD [5].

However, HDD transfer rate does not grow proportionally to the storage capacity, as displayed in Figure 1.1 and Figure 1.2. While CPU and RAM technology continues to advance, the HDD has become the main limiting factor of the

1. The CPU cache is memory located on the CPU chip, and is based on static RAM (SRAM)—a faster chip than the dynamic RAM (DRAM) chip used in main memory. The CPU cache is used to reduce the average access time to data in the main memory.

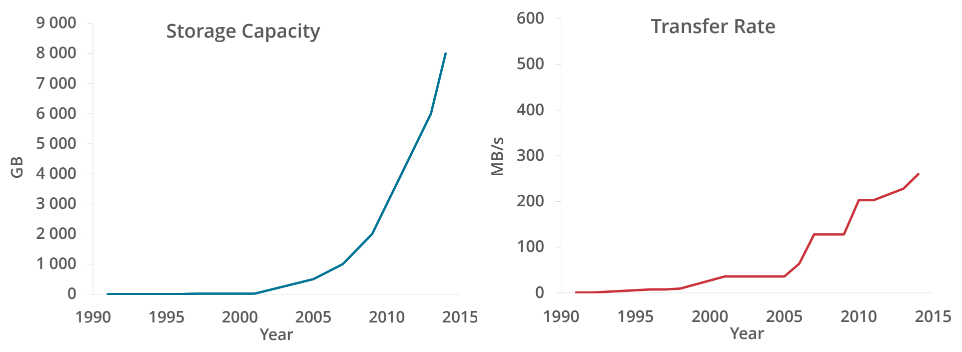


Figure 1.1: HDD storage capacity and transfer rate development. Sources: [2], [4], [6], [7].

overall performance of many systems, which has lead to the development of solid state drives (SSDs). An SSD outperforms a mechanical storage device by orders of magnitude, both in terms of I/O operations per second (IOPS) and transfer rate, but current interfaces are unfortunately not able to fully exploit SSDs.

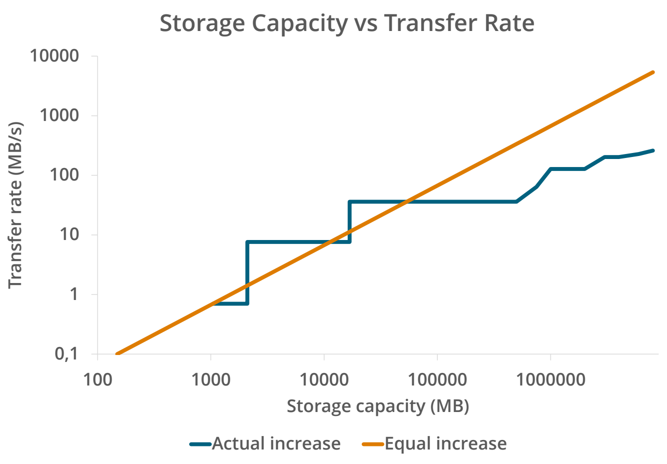


Figure 1.2: Comparison between the increase in HDD storage capacity and transfer rate. The straight line represents an ideal growth in both capacity and transfer rate; that is, the ratio between the two do not change. The jagged line represents the actual change in ratio between the two. This line shows that at two occasions the transfer rate increased more than capacity, but that the ratio has shifted and continued to favor an increase in capacity. Note that the axes are in $\log_{10}$ scale.

Serial ATA (SATA), Small Computer System Interface (SCSI), and Serial Attached SCSI (SAS), have been, and still are, the main technologies used for attaching storage devices to a host computer system. However, these technologies are storage-oriented, and their designs are permeated by the assumption that the connected device is mechanical. For example, current SATA-technology does not match the capabilities of an SSD, effectively limiting SATA-connected SSDs to less than 6 Gbit/s. Exploiting SSD performance within the constraints of a traditional bus attachment is difficult, and attachments through PCI Express (PCIe) are gaining adoption. SSDs using this type of interconnect are already on the market, and are currently offering transfer rates beyond 24 Gbit/s [8].

1.1 Non Volatile Memory Express

Non-Volatile Memory Express (NVMe) [9] is a recently developed standard for operating SSDs that are connected to a host system through PCIe. The interface is designed to address the needs of enterprise and consumer systems, and provides optimized command submission and completion paths. With features such as support for parallel, priority based operation of up to 65 535 input/output (I/O) queues, namespace management, and advanced power management options, NVMe allows us to rethink how we interact with storage devices.

Namespaces are NVMe's equivalent to a logical unit (LUN), and are isolated from each other. Each namespace can have its own set of I/O queues, and can be configured individually. A namespace also supports variable block sizes, and per-block metadata for storing protection information or other information. The number and size of I/O queues, and the sequence in which these are processed, is configurable. For example, the controller may use a round robin (RR) arbitration to launch one command at a time from each active queue, as shown in Figure 1.3, or in bursts.[2]

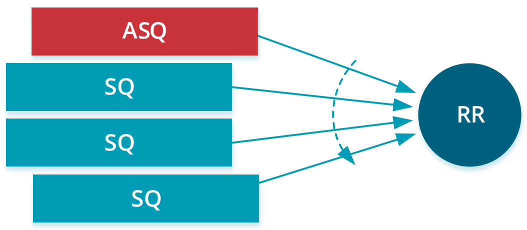


Figure 1.3: With RR arbitration, every submission queue is treated equally, including the admin queue. The controller can be configured to process a single command from each queue at a time, or in bursts.

Additionally, weighted round robin (WRR) arbitration may be used, where each I/O queue is assigned a priority. Using this setting, the controller will,

2. NVMe supports burst rates of 2, 4, 8, 16, 32, 64, or limitless, the latter means that all commands present in a single queue are immediately processed.

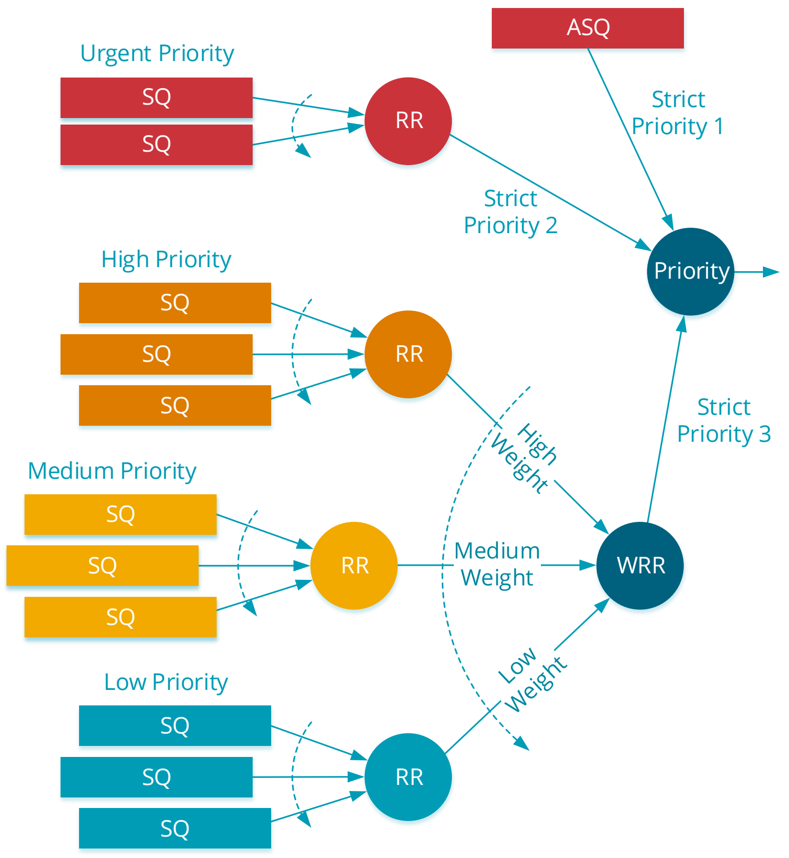


Figure 1.4: WRR arbitration consists of three priority classes and three WRR priority levels. The *admin* class is assigned to the admin submission queue, and is prioritized over any other command. The *urgent* class is only outranked by the admin class, and might starve the lower classes if not used with caution. The lowest priority class, the *weighted round robin* class, consists of three priority levels that share the remaining bandwidth. Each WRR priority group is arbitrated internally in a RR fashion.

based on a defined metric, launch commands with respect to the priority of each queue. The WRR arbitration method, illustrated in Figure 1.4, consists of three classes [9]:

- **Admin**, which applies to the admin queue. Any command submitted to a queue of this class, such as an abort, is prioritized above all other commands that has been or is yet to be submitted to a queue of a different class.

- **Urgent**, which is similar to, but ranked below, the admin class. Commands submitted to an urgent class queue are immediately processed, unless the admin class has outstanding commands.

- **Weighted Round Robin**, which is the lowest prioty class, consists of three priority levels that share the remaining bandwidth. The priority levels *high*, *medium*, and *low* are scheduled with WRR arbitration based on weights assigned by host software.

1.2 Vortex

Cloud environments often employ virtual machines (VMs) that rely on a virtual machine monitor (VMM) to schedule physical resources. The virtualized environment allows multiple instances of operating systems (OSs) to co-exist on the same system, improving utilization of the physical machines. Service providers and customers establish a service level agreement (SLA): a contract that states the amount of resources that shall be available to the customer at all times. The VMM must therefore schedule the available resources in a manner that honors any active SLA, regardless of how many VMs that are present. These agreements cover anything from CPU time and available memory, to network bandwidth and storage capacity.

The omni-kernel architecture (OKA) is a novel OS architecture designed to offer strong isolation and accurate, fine grained scheduling of system resources for all tenants that share a platform [10], [11]. The OKA is divided into resources, which provide access to both hardware and software components, and uses messages to communicate between them. By controlling the flow of messages, the OKA ensures that all resource consumption resulting from a scheduling decision is measured and attributable to an activity.[3]

Vortex is an OS kernel implementing the OKA, and offers pervasive monitoring and scheduling of resources at a typical cost of 5 % CPU utilization or less [10]. Unlike conventional VMMs [12], [13], Vortex does not offer virtual device interfaces to its VMs; rather, high-level abstractions and features are presented to a guest OS, targeting compatibility at the application level. Functionality and abstractions offered by Vortex—multithreading, networking, processes, memory management, and files—facilitate the implementation of a thin guest OS that translates native Vortex system calls to the system call interface of the OS it virtualizes. An example is a minimal port of Linux, on which unmodified applications such as Apache [14], MySQL [15], and Hadoop [16] can be run [17], [18].

The OKA's focus on fine-grained scheduling and performance isolation makes

3. The OKA defines an activity as any unit of execution, for example a process, a VM, or a service.

the features of NVMe particularly desirable. For example, the option to customize and associate priority with a queue, to adjust the burst rate of each queue, or even to assign a varying number of queues to different processes, can help better uphold the resource shares assigned to different activities. The ability to isolate portions of a disk using namespaces, and assigning queues private to each namespace, is also very attractive in a virtualized environment.

This thesis describes the implementation of an NVMe driver for Vortex that is exposed as a SCSI device, such that the already implemented storage stack can remain untouched. We explore what impact adding support for NVMe will have on the existing system, and whether Vortex is capable of supporting this new kind of storage device.

1.3 Problem Definition

This thesis investigates whether Vortex can exploit the recently introduced NVMe technology. A particular focus and goal is to identify shortcomings and opportunities for improvement if the current Vortex storage stack is to host an NVMe device. To give additional weight to findings, the approach will be experimental, aiming to implement a working NVMe driver.

1.4 Scope and Limitations

The NVMe specification defines support for features such as end-to-end data protection, per-block metadata, namespace sharing and multipath I/O [9], which may be very useful when building large systems. The center of interest in this thesis, however, lies in the exploration of whether an already implemented system is able to support and make use of a storage interface that features a large number of data pathways, contrasting with the single pathway of conventional interfaces.

In our experiments, we use an Intel DC P3600 SSD, and we are in general limited by the capabilities of this disk. The disk includes a fixed namespace, RR arbitration with a single command burst, support for 31 I/O queues (IOQs), and a maximum queue depth of 4096 entries. We have focused on this set of supported features.

1.5 Methodology

The final report of the ACM Task Force on the Core of Computer science states that the discipline of computing consists of three major paradigms: theory, abstraction, and design [19]. Albeit intricately intertwined, the three paradigms are distinct from one another in the sense that they represent separate areas of competence:

Theory is the search for patterns. With mathematics as the methodological paradigm of theoretical science, the patterns are used to form conjectures that are verified or disproved by mathematical proof [20]. The process consists of a characherization of objects, hypothesizing possible relationships among them, proving whether they are true, and an interpretation of the results. These steps are expected to be iterated when errors and inconcistencies are discovered.

Abstraction is rooted in the experimental scientific method. The approach uses relationships between objects to formulate predictions that can be compared with the world. A hypothesis is tested by creating a model and designing experiments, which are used to collect data. The produced data is used to either verify or disprove the hypothesis. When results contradict the prediction, an engineer expects to iterate the steps.

Design is the bedrock of engineering, and uses the implementation of specific instances of relationships to perform useful actions. In the process of constructing a system, requirements and specifications form the grounds for its design and implementation. Once implemented, the system is tested to reveal whether the implementation is satisfactory. The steps are expected to be iterated until the requirements are met.

The nature of this thesis is in systems research, investigating whether an existing system is capable of accommodating technology that changes the way storage is interfaced. We construct a hypothesis on whether a current system is able, with few modifications, to accommodate and exploit technology that differs from what is currently supported. We incorporate the new technology in our system and design experiments that measure and compare the achieved performance with its predecessor.

As part of a systems research project, we do not aim to meet a finite set of requirements, but to continuously iterate the design process and use the results to explore and compose requirements for a new and better version.

1.6 Contributions

This thesis makes the following contributions:

- Vortex currently implements support for storage through the SCSI interface. This project expands the storage stack with support for NVMe-capable storage.

- The Vortex device system has been altered to enable configuration of pin-based interrupts for devices that do not appear in the MP-table, or have failed to be configured due to PnP errors.

- We add support for configuration of devices that require initialization over pin-based interrupts before activating more advanced interrupt delivery.

- The NVMe driver is implemented with the possibility to customize how commands are issued to the storage device. We evaluate each implemented method.

- We measure and evaluate the storage performance gain from employing PCIe-based SSDs in Vortex, and discuss other possible benefits of using a multi queued storage interface.

1.7 Outline

The rest of this thesis is organized as follows:

Chapter 2 describes the development in how we interface storage, and the evolution of the interconnect that supports them.

Chapter 3 presents the improvements made that allow Vortex to configure devices that are unrecognized or not supported by the BIOS.

Chapter 4 describes how we expose an NVMe SSD as a SCSI device in Vortex, such that these devices may be used without changing the current storage stack. We also present the improvements made that allow us to change interrupt delivery method for a configured device.

Chapter 5 evaluates the implementation by measuring the achieved throughput when performing reads and writes to an NVMe device.

Chapter 6 discusses future work and concludes the thesis.

2 Background

While the rest of the computer components are generally becoming more parallelized, storage devices have achieved higher transfer rates by changing from a parallel to a serial interface. This chapter starts by describing the advantage of using a serial interface rather than a parallel one, and move on to presenting a historical lineage of the interfaces that have had the most impact in both enterprise and consumer markets. Following that is a description of the advances that have lead to the development of NVMe. Finally, we present related work that also aims to better utilize multi-queued interfaces.

2.1 From Parallel to Serial Communication

Improvement in storage interfaces cannot be exploited if the underlying connection to the host system is slow or incompatible with a device. As mentioned in Chapter 1, interconnect technology is diverging from the development pattern in other components of the computer by migrating data transfer from parallel to serial communication.

Normally, serial communication is slower than parallel. For example, a 32 bit wide parallel bus is able to transfer 32 times as much data per clock cycle than a serial bus which only transfers a single bit per cycle. The parallel bus is, however, prone to clock skew issues and electromagnetic interference (EMI), especially when the clock rate increases [21].

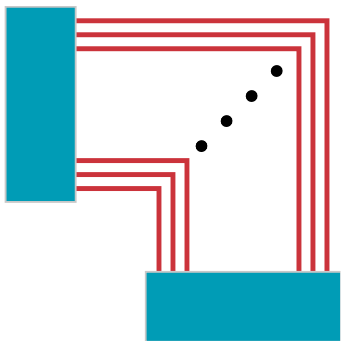


Figure 2.1: Parallel buses transfer 32 to 64 bits per clock cycle, but are prone to interference and variances in propagation delay, due to varying length of the wires.

Skew

In circuit designs, skew is the time delta between the actual and expected arrival time of a clock signal. When designing a bus, for example the 32 bit parallel Peripheral Component Interconnect (PCI)-104 bus [22], it is difficult to ensure that all 104 wires are of equal length. Figure 2.1 illustrates a parallel bus that includes a 90° angle that causes the outer wires to be longer than the innermost wire. Because no two lines are equal in length, the *propagation delay*[1] will be different. For short distances and slow clock rates, this is not an issue, but as the distance and signaling frequency increases, the difference becomes significant. For example, the shortest wires may be able to deliver a second bit before the longest wire delivers the first bit, violating the integrity of the transmitted data.

Electromagnetic Interference

Ampère's law states that if a current flows through a conductor, a magnetic field appears around it, and that the strength of the magnetic field is proportional to the net current that passes through the conductor [23]. It is, for example, because of this law that we are able to create powerful electromagnets.

In printed circuit boards (PCBs), this creates an unwanted effect, called *crosstalk*: a coupling of energy between transmission lines that causes signals to appear on both lines when they are only desired on one [24]. As mentioned, for low-powered parallel buses this is not a big problem, but when the frequency of the bus is increased, the magnetic forces increase, compromising signal integrity.

1. The propagation delay is the time taken to transport a signal from source to destination.

2.1.1 Evolution

The parallel PCI bus has until recent years been the main interface for connecting peripherals to a computer, and was, until 2004, able to keep up with the increasing requirements for throughput. The original 32 bit PCI local bus standard, or *legacy PCI*, operated at 33 MHz, supporting a transfer rate of 133 MB/s [22]. Peripheral Component Interconnect eXtended (PCI-X) is an enhancement of the PCI bus, and increased the throughput to 4.2 GB/s [25].

PCI-X 3.0 was created as a last improvement in 2004. It defines an operational speed of 1066 MHz, resulting in a transfer rate of 8.5 GB/s. However, Intel started sidelining PCI-X in favor of the serial PCIe interface, which requires only a fraction of the transmission lines of a parallel interface, avoiding the "1000 pin apocalypse" [26].[2]

PCIe is a high-speed serial bus, and replaces the parallel PCI and PCI-X standards. Additionally, PCIe supports native hot swap[3] functionality and Advanced Error Reporting (AER). Recent versions also implement I/O virtualization, a technology that allows multiple VMs to natively share PCIe devices [27]. PCIe requires only four wires to transmit data, in unison called a *lane*, two for transmitting and two for receiving, as shown in Figure 2.2.

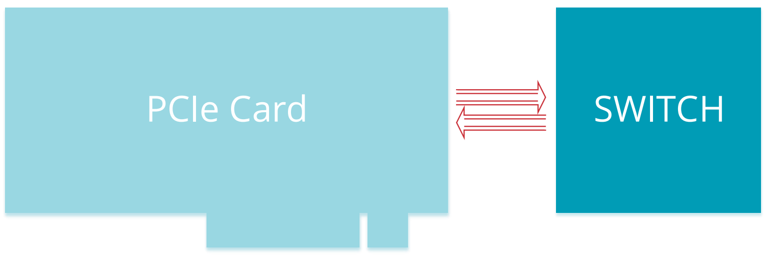


Figure 2.2: A PCIe x1 link uses four wires and transmits one bit in each direction per cycle.

The bus is not vulnerable to skew since only one bit is transmitted per direction per clock cycle. PCIe also avoids EMI by employing a technique called *differential signaling*, in which the same signal is transmitted simultaneously on two wires, but with an inverted polarity on the second wire [28], as depicted in figure Figure 2.3. Thus, PCIe is able to operate at frequencies of 8.0 GHz, or a transfer rate of 8 gigatransfers per second (GT/s) [29]. Multiple lanes are often used to further increase the bandwidth between the device and the host system. Figure 2.4 illustrates an x4 link, which supports roughly four times the

2. A 32 bit PCI-X 2.0 connector contains 184 pins.
3. *Hot swapping* is a term used to describe the action of removing or replacing a component without having to power down the system.

transfer speed of an x1 link.

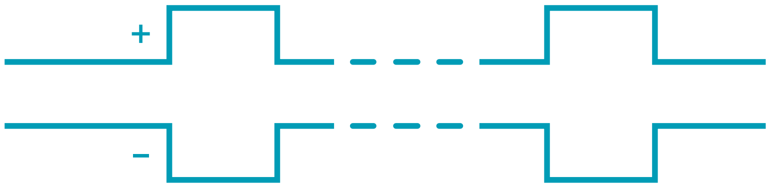


Figure 2.3: Differential signaling provides a higher immunity to EMI. With this technique, a single signal is transmitted on two adjacent wires, with the second wire transmitting the signal "mirrored" to the first, canceling out the interference.

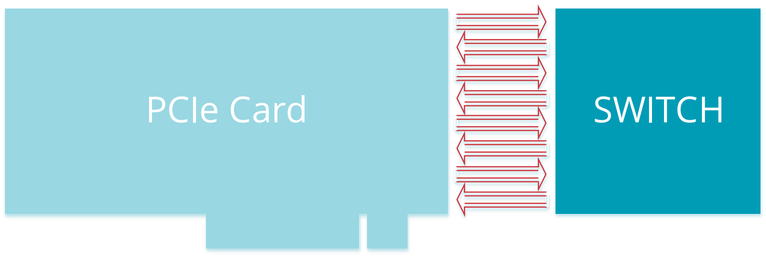


Figure 2.4: A PCIe x4 link uses sixteen wires and transmits four bits in each direction per cycle.

Data Encoding

Although the number of transferred bits per second is equal to the operating frequency of the bus, the actual usable data transferred is less. The serial PCIe bus does not use dedicated wires to transmit clock signals, but relies on a frequent transition between ones and zeros in the data stream to recover the clock period. A deterministic transition pattern is ensured by employing an encoding scheme in which extra bits are used to limit the number of consecutive ones or zeros [28].

$$\text{bandwidth} = 2R \times \text{lanes} \times \frac{\text{bits}}{\text{line code}}$$

Equation 1: To calculate the aggregate bandwidth of a PCIe link, multiply the link's bitrate R by 2 to account for the dual-simplex link, and multiply the product by the number of lanes in the link. Finally, multiply the result by the encoding overhead to get the number of bits of actual data that is transferred per second.

PCIe 2.0 uses an 8b/10b encoding, and uses ten bits to transfer a byte—a 20 % overhead—while PCIe 3.0 reduces the overhead to approximately 1.54 %

with a 128b/130b encoding. From the formula in Equation 1, we see that for a PCIe 3.0 device using an x1 connection, the actual bandwidth is approximately 1.97 GB/s or 985 MB/s in each direction.

2.2 Interface Lineage

Many different solutions to interfacing a storage device have emerged over the years. Interfaces such as the FD-400 8 inch floppy disk driver, the Enhanced Small Device Interface (ESDI), and most proprietary interfaces, have not survived, while others, such as Integrated Drive Electronics (IDE) and SCSI based interfaces, have remained and are still widely used. Here follows a description of the interfaces that have stood the test of time, and are still in use.

2.2.1 Small Computer System Interface

Small Computer System Interface (SCSI), originally named Shugart Associates Systems Interface (SASI), was developed in the years 1978–1981 by the Shugart Associates Company, who based it on the *Selector Channel* in IBM-360 computers. In 1986, only few years after being publicly disclosed in 1981, SASI became an ANSI standard and the name was changed to SCSI [30]. The SCSI standard defines how computers physically connect with and transfer data to and from peripheral devices. The uses of SCSI range from Compact Disk (CD) drives and printers to HDDs, where the latter is the most common.

SCSI uses a *bus topology*, meaning that all devices are daisy-chained linearly, as depicted in Figure 2.5. Each bus supports up to eight devices, but expanders can be used to allow more SCSI segments to be added to a SCSI domain. A SCSI bus must be terminated at the end, such that the bus appears electrically as if it is infinite in length. Any signals sent along the bus appear to all devices and end in the terminator, which cancels them out, such that there are no signal reflections that cause interference [31], as shown in Figure 2.6.

2.2.2 Integrated Drive Electronics / Parallel-ATA

Integrated Drive Electronics (IDE), also referred to as Parallel ATA (P-ATA), is the the result of further development of the Industry Standard Architecture (ISA) interface developed for use in IBM's Personal Computer AT (PC/AT)—a bus that supported a parallel transmission of 16 bits at a time. The IDE channel was designed as a pure HDD interface since other proprietary interfaces already existed for devices such as CD-ROMs and tape drives. However, during

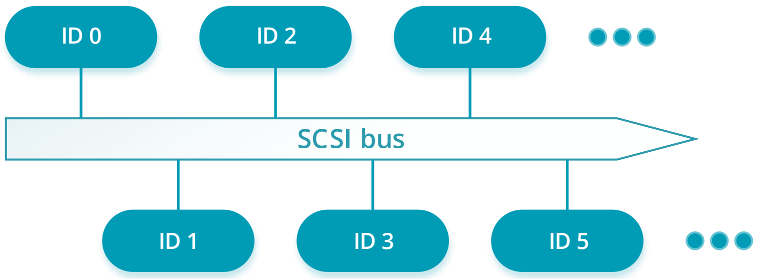


Figure 2.5: The SCSI bus is a linear daisy chain of up to eight devices, but expanders allow more SCSI segments to be added to the domain.

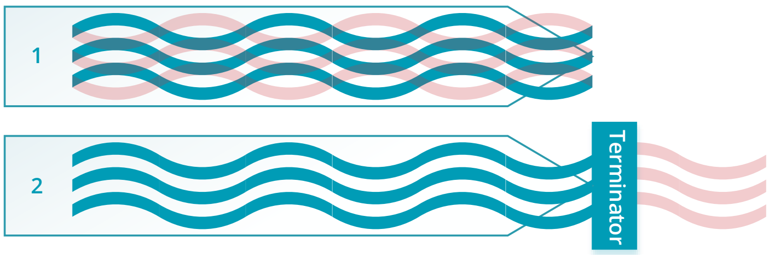


Figure 2.6: A SCSI bus must be terminated to avoid interference. A terminator stops the signals at the end of the line, and makes it appear as if the bus is infinite in length. The figure illustrates an unterminated bus (1) and a terminated bus (2). The terminated bus appears to be infinite in length, and avoids interference from signals that bounce back from the end.

the 1990's, it became obvious that a single, standardized interface would be preferable to the proprietary interfaces.

Because the AT Attachment (ATA) command structure is incompatible with anything but HDDs, the AT Attachment Packet Interface (ATAPI) was developed to work on top of IDE [32], which allows the ATA interface to carry SCSI commands and responses. ATAPI became very successful, and is still used in modern SATA interfaces.

Programmed I/O

Early versions of ATA operated in programmed I/O (PIO) data transfer mode, which occurs when the CPU instructs access to a device's I/O space for data transfer. PIO is relatively simple and cheap to implement in hardware, but has the disadvantage that the CPU is responsible for all data transfer, as illustrated in Figure 2.7. This means that the CPU consumption increases proportionally with the transfer rate, potentially creating a bottleneck in the overall computer performance.

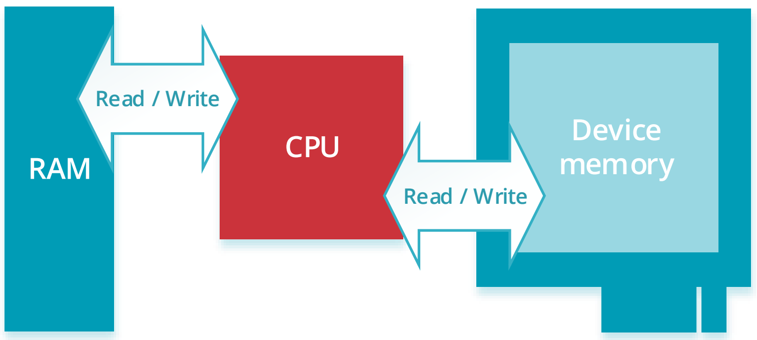


Figure 2.7: Programmed I/O occurs when the CPU instructs access to a device's I/O space for data transfer. As the transfer speed increases, the CPU resource consumption increases as well.

In modern systems, PIO has been replaced with direct memory access (DMA), but is still implemented in interfaces that do not require high transfer rates, including serial ports, and the PS/2 keyboard and mouse ports.

Direct Memory Access

Unlike PIO, where the CPU controls and monitors data transfers to and from a peripheral device, the device operating in DMA mode is programmed to perform data transfers to and from host memory on behalf of the CPU, as depicted in Figure 2.8. The only interaction required by the CPU is to grant the controller access to the system bus for data transfer.

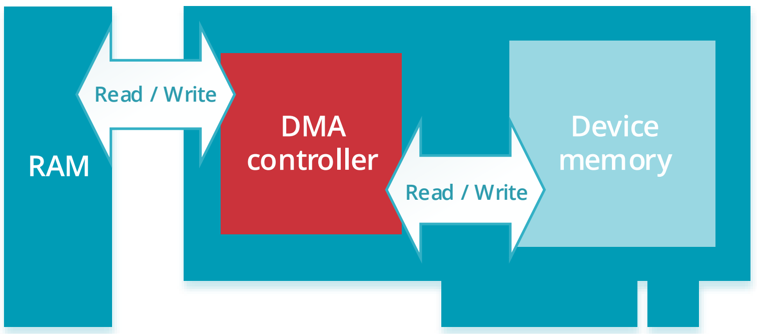


Figure 2.8: Direct memory access allows a peripheral on the system bus to perform reads and writes to host memory on behalf of the CPU. The CPU is free to perform other tasks while data transfer is performed, and is notified by the device via interrupts when the transfer is complete.

Modern PCIe devices can be configured as *bus masters*, allowing the DMA controller to initiate transactions without involvement from the CPU. While data transfers are handled by the controller, the CPU is free to perform other tasks, and may be notified of any changes in the memory area governed by the peripheral through interrupts [33].

2.2.3 Serial-ATA

Serial ATA (SATA)-600 is the result of a continuous effort to improve an existing interface, and offers a theoretical maximum speed of 600 MB/s, while retaining backwards compatibility with earlier versions of the interface, such as SATA-300 [34]. Besides supporting high speed devices, SATA also supports hot swapping. Redundant storage systems benefit from this ability, as a faulty HDD may be replaced without having to disconnect the service.

Advanced Host Controller Interface

Advanced Host Controller Interface (AHCI) is an application programming interface (API) that defines a mode of operation for SATA. The AHCI device is a PCI class device that acts as a data movement engine between system memory and SATA devices, providing a standard method of interaction between the host system and SATA devices. This simplifies both detection, configuration, and programming of SATA and AHCI adapters [35]. An AHCI device, or host bus adapter (HBA), is required to be backwards compatible with ATA and ATAPI compliant devices, as well as both the PIO and DMA protocols.

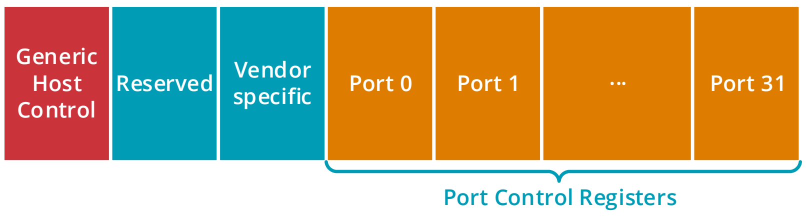


Figure 2.9: AHCI HBA memory consists of *Generic Host Control* registers that control the behavior of the entire controller. *Port Control* registers contain information for each port, such as two descriptions per port, which are used to convey data.

The system memory structure described by AHCI contains a generic control and status area, a pointer to a descriptor table used for data transfers, and a command list, in which each entry contains the information necessary to program a SATA device. Figure 2.9 shows a simplification of this memory structure. In addition to implementing native SATA features, AHCI specifies the support for 1 to 32 *ports*, to which a SATA device can be connected. The AHCI ports support simultaneous transmission of 32 commands.

2.2.4 Serial Attached SCSI

Like SATA is an improvement of P-ATA, and has become widespread in the consumer and small business market, Serial Attached SCSI (SAS) builds on, and replaces the older parallel SCSI interface with a serial point-to-point protocol. SAS has for a long time been the main choice in building enterprise storage systems, and is usually the choice of interface if performance and reliability is of concern. Being a more costly system, it is used almost exclusively in medium and large systems [36].

SAS supports a greater number of devices than the original SCSI interface—up to 65 535 devices if expanders are used—and throughput up to and past 1200 MB/s [37]. Like SATA, SAS devices also support hot swapping, a highly desirable feature in large data centers where disk-failures are the norm rather than the exception. Furthermore, some SAS sockets are designed for compatibility with SATA devices.[4]

2.3 From Magnetic to Flash Based Storage

With the advances in storage interface technology, HDDs are no longer able to keep up. In a mechanical HDD, a motor rotates the platters while an actuator arm moves the heads across the magnetic surface to read or write data. The moving parts cause turbulence, which becomes problematic when the rotation speed increases. Current high end HDDs are therefore limited to 15 000 rpm and 10 000 rpm, producing an average seek time[5] below 3.0 ms [38], [39].

Because of the limited performance, but low cost, of HDDs, the interest in redundant arrays of independent disks (RAIDs) exploded [40]. RAID 0, or striping, does not actually offer redundancy, but spreads the data evenly across the entire array of disks, which in return produces a higher throughput. RAID 0 is, for example, widely used in supercomputing environments where performance and capacity are the primary concerns. Other RAID levels are designed to provide fault tolerance by mirroring the written data on two or more drives (RAID 1), or by using dedicated parity drives that can be used to recreate corrupted information (RAID 2–6).

In recent years, the price of not and (NAND) flash memory has decreased

4. Some SAS sockets are designed such that SATA devices may be uses as well, but SATA sockets do not support SAS devices.
5. *Seek time* is the accumulated time taken for the actuator arm to move the heads to the track where data is to be read or written, and the time it takes for the platters to rotate such that the data blocks are positioned under the heads.

drastically, and SSDs are becoming popular in systems that require better disk performance than HDDs can offer. SSDs have no moving parts, and are primarily composed of NAND flash memory. Perhaps the most important component in the SSD is the Flash Translation Layer (FTL) that translates the logical block addresses (LBAs) of the host to its corresponding physical block addresses (PBAs) in the storage device, so that an OS can read and write to NAND flash memory like it would with disk drives [41]. The FTL is the SSD's equivalent to the actuator arm in an HDD, but offers, in contrast, a more or less constant access latency of between 20 µs and 120 µs across the entire drive [42]. This corresponds to an access time speedup factor of 150 when compared with an HDD.

2.3.1 Non-Volatile Memory Express

In contrast to SATA and SCSI, NVMe has been architected from the ground up, and targets flash-based storage. NVMe devices are directly attached to the CPU subsystem through PCIe, and offer a high level of parallelism and reduced latency. This has improved both random and sequential performance [43]. The interface supports up to 65 536 I/O queue pairs, each large enough to support up to 65 536 outstanding commands—2048 times the number of supported ports and commands of AHCI.

NVMe is a standard-based initiative by an industry consortium consisting of more than 80 large companies,[6] to develop a common interface for connecting high-performance storage. Version 1.0 of the NVMe interface was released on March 1st 2011, which defines the queuing interface, NVM and Admin command sets, end-to-end protection, and physical region pages.

The NVM command set is designed for simplicity: every command is 64 bytes, which is sufficient for a read or write of 4 KB. For a virtualized environment, the namespace system offers isolation between LUNs, with I/O queues private to a namespace, and while AHCI requires 4 uncacheable register reads, each translating to about 2000 CPU cycles, NVMe requires none.

6. The NVM Express Work Group includes companies such as Intel, CISCO, DELL, and Samsung.

2.4 Related Work

The Serial ATA International Organization[7] has, based on SATA, designed an improved interface for targeting PCIe connected SSDs [44], with backwards compatibility for conventional SATA devices [45]. The interface consists of two SATA 3.0 ports, and a PCIe x2 connector, which allows Serial ATA Express (SATA Express) to supports both AHCI for software level backwards compatibility with SATA devices, and NVMe devices. The interface supports the use of either SATA or PCIe, but not both in tandem, setting a maximum theoretical transfer rate of 1.97 GB/s.

SCSI Express is a more direct competitor to NVMe, and targets an interface that carries SCSI commands over PCIe. The interface uses two T10 standards, SCSI over PCIe (SOP) and a PCIe Queueing Interface (PQI), and operates similarly to NVMe, with one or more pairs of inbound and outbound queues [46]. A goal of SCSI Express is to be flexible, and to support SAS, SATA, SATA Express, SCSI Express, and NVMe devices.

In the field of networking, multi-queued interfaces have existed for some time, but have, like storage interfaces, continued to use well established APIs, such as Berkeley Sockets. NetSlices [47] provides an OS abstraction that enables a more efficient utilization of multi-queue network interface card (NIC). The NetSlice abstraction enables linear performance scaling with the number of cores in the system while processing network packets in user-space. This work shows that multiprocessor systems benefit from interfaces that allow parallel operation of multiple queues, but also that changes in the API are required to fully exploit newer technology.

7. `http://sata-io.org/`

3 Device Configuration

The Basic Input Output System (BIOS) is normally able to recognize and perform basic configuration of a device connected to the PCI subsystem, and store interrupt routing information in the MultiProcessor (MP) table such that an OS can retrieve the information through a simple lookup. If the BIOS fails to prepare the device, the basic information is not stored, and must be gathered by the OS if the device is to be used.

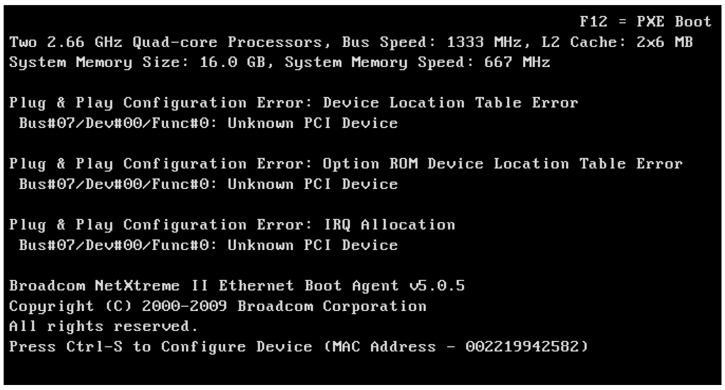
One shortcoming that we identified with the current Vortex implementation, was the reliance on BIOS-supplied information and configuration. Failure by the BIOS to identify or configure an attached device results in Vortex not being able to use that device. The BIOSs of the hardware platform currently supported by Vortex does not recognize NVMe devices. To remedy this problem, a contribution of this thesis is the design and implementation of a system for configuration of PCI bridges in Vortex. This system enables Vortex to recognize host-attached NVMe devices and, crucially, to correctly configure device interrupt management. This includes the NVMe-particular need to initially operate the device using conventional pin-based interrupts, before changing to the more modern message signaled interrupts (MSI) or MSI extended (MSI-X) after performing the first steps of device configuration.

In this chapter, we first present the mechanism used by the BIOS to execute basic configuration, and causes of failure. Thereafter, we describe how data collected from PCI bridges can be used to determine the missing information. We then describe the implementation of a PCI-to-PCI bridge device driver that is used to assist in the mapping of the PCI hierarchy, and how we use this mapping to retrieve the information needed to configure pin-based interrupts for an NVMe device. We end the chapter with a short summary.

3.1 Basic Device Discovery and Configuration

The large variety of available devices—graphics processing units (GPUs), network adapters, and storage devices—allows us to tailor and improve computers to better suit our needs. However, devices require resources to work, and conflicts may occur when more than one device is attached to the same computer. Plug and Play (PnP) is designed to let hardware and software work together to automatically configure devices and allocate resources [48], rather than requiring a user to perform complicated setup procedures for every component. But for PnP to work, both the host system and the attached peripheral device must be capable of responding to identification requests, and accept resource assignments, and the BIOS must collect and communicate information about devices to the OS. Additionally, the OS must set up drivers and other necessary low-level software to enable applications to access the device.

If these requirements are not met, PnP configuration fails. An example of this is when the BIOS has insufficient information to recognize all types of devices, resulting in a PnP configuration error, effectively hindering the computer from using the device. Figure 3.1 displays a screenshot from a DELL PowerEdge M600 blade server with such a problem. The server is equipped with an unrecognized device, an NVMe SSD, and is unable to configure it.



```
                                                                F12 = PXE Boot
Two 2.66 GHz Quad-core Processors, Bus Speed: 1333 MHz, L2 Cache: 2x6 MB
System Memory Size: 16.0 GB, System Memory Speed: 667 MHz

Plug & Play Configuration Error: Device Location Table Error
 Bus#07/Dev#00/Func#0: Unknown PCI Device

Plug & Play Configuration Error: Option ROM Device Location Table Error
 Bus#07/Dev#00/Func#0: Unknown PCI Device

Plug & Play Configuration Error: IRQ Allocation
 Bus#07/Dev#00/Func#0: Unknown PCI Device

Broadcom NetXtreme II Ethernet Boot Agent v5.0.5
Copyright (C) 2000-2009 Broadcom Corporation
All rights reserved.
Press Ctrl-S to Configure Device (MAC Address - 002219942582)
```

Figure 3.1: PnP Configuration Error.

An OS is much more flexible than the BIOS when it comes to configuring a device, and can usually configure even those that failed during boot. In our case, Vortex is able to discover the SSD and read its capabilities, and attempts to configure it. However, NVMe requires that initialization is done over pin-based interrupts [9], and the aforementioned PnP error causes the BIOS to ignore the device, which means that when Vortex queries the BIOS for interrupt request (IRQ) information for the device, it is not available.

3.2 Configuring Devices Present on a Secondary Bus

In a PCI hierarchy, a bridge is an endpoint that provides a connection path between two independent PCI buses [49], as illustrated in Figure 3.2. The primary function of a bridge is to allow transactions between a master on one bus, and a target on the second bus. Devices that reside on a secondary bus may have their IRQ information determined from the hardware address of the parent bridge.

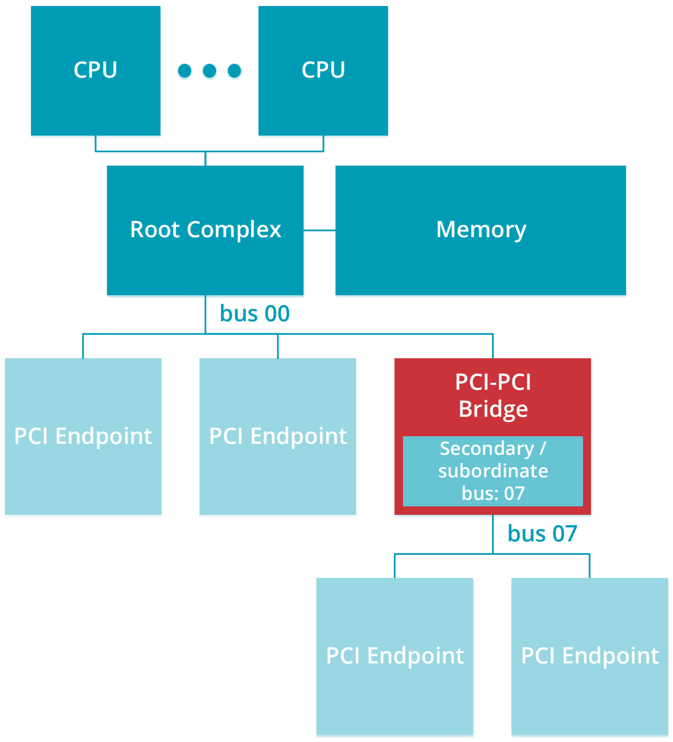


Figure 3.2: The PCI-to-PCI bridge connects the processor and memory subsystem to the PCI switch fabric composed of one or more switch devices. The bridge device implements PCI header type 1, which includes the *secondary* and *subordinate* bus number registers. When combined, these registers define the range of buses that exists on the downstream side of the bridge [28].

Until now, Vortex has worked under the assumption that all devices are successfully detected during boot, and that IRQ information is immediately available for any device. IRQ information for pin-based interrupts is usually obtained from the MP table. The MP table was introduced along with the Advanced Programmable Interrupt Controller (APIC), and enumerates the processors and APICs in a computer, as well as describing the routing of PCI interrupts to APIC input pins [50]. The MP table is, however, not guaranteed to include information about the entire system. The error shown in Figure 3.1 affects the configuration of a device that resides on an expansion bus, and the MP table will typically not list buses behind a bridge.

During device discovery, Vortex currently ignores anything but network and storage class devices. But to obtain the missing IRQ information for our device, we require a mechanism that is able to discover the PCI hierarchy.

3.3 Vortex Class Drivers

In addition to low level device drivers, Vortex implements *class drivers*—a high level abstraction that handles all instances of a device type, such as storage and network. Each class driver implements a *class multiplexor*: a set of callback functions that allows the device subsystem to use the same interface when accessing different classes. The classmux, displayed in Code Snippet 3.1, defines functions used by the kernel to start and stop a device, and to get or set its current state.

Code Snippet 3.1: The Class Driver Multiplexor standardizes how the device subsystem communicates with different device drivers.

```
struct devclassmux_t {
    dcl_new_t           dcl_new;           // Instantiate new classmux type
    dcl_get_id_t        dcl_get_id;        // Get device identifier
    dcl_get_cap_t       dcl_get_cap;       // Get device capabilities
    dcl_start_t         dcl_start;         // Start device
    dcl_stop_t          dcl_stop;          // Stop device
    dcl_write_done_t    dcl_write_done;    // Action on completed write request
    dcl_devbuf_alloc_t  dcl_devbuf_alloc;  // Allocate device specific buffer
    dcl_devbuf_free_t   dcl_devbuf_free;   // Free device specific buffer
    dcl_isoperational_t dcl_isoperational; // Set or read device operational state
};
```

Devices that should be accessible from kernel or userland processes also specify a resource interface that maps request types to their respective functions in the class driver. The resource interface depicted in Figure 3.3 is common for all I/O devices in Vortex, and allows a process to access any device without changing anything but the destination of the request.

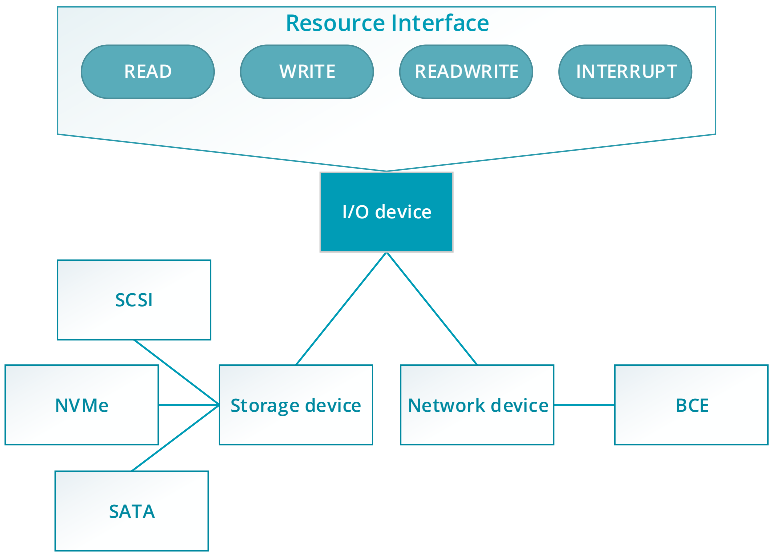


Figure 3.3: Vortex class drivers may export a resource interface that allows processes to interact with any device by using the same functions. Requests are multiplexed and routed to a device class driver, which invokes the corresponding device driver that translates the request to a device specific command. Thus, a process only needs to change the resource path to interact with a different device type.

To support the detection of the PCI bus hierarchy, Vortex must be able to reference the bridges that expand to secondary buses. The kernel requires that devices are associated with a class driver, and that the class driver is able to set and report the operational state of the device.

The implemented bridge class driver is a minimal implementation, and supports the functionality necessary to instantiate a new bridge device type, and to set or report the operational state of a device. This state is only logical, and we ignore all commands that would change the state of the hardware.

3.4 PCI-to-PCI Bridge Device Driver

Although the class driver abstraction offers simplicity in process-to-device interaction, specialized device drivers are still required to communicate with the actual hardware. The PCI-to-PCI bridge driver allows PCI-to-PCI bridges that master secondary buses to be instantiated as Vortex device objects and be added to the global device list.

Because we want bridges to be passive, and to let interrupts be produced by devices located on the secondary bus, rather than the bridge, the bridge driver reports the NOINT capability. This capability stops the kernel from configuring interrupts for the given device, regardless of the capabilities listed in the device's PCI header. Devices present on the secondary bus are not affected by this.

3.5 Configuring a Parent Bridge

During system initialization, the pci_probe function enumerates all peripherals connected to the PCI bus by probing the PCI BIOS for all known devices. Code Snippet 3.2 shows the linear search through the PCI subsystem. The probe queries all permutations of device class and location in the subsystem.

Code Snippet 3.2: Probing of PCI devices. If the PCI BIOS finds a device, it is matched with the types supported by the system, and either configured, or ignored.

```
for class in pci classes {
    for index in range 0 to 255 {
        // Probe the PCI BIOS for the class at index
        busdevfunc = asm_pci_find_class(class, index);
        if (busdevfunc == -1)
            break;

        switch class {
        case NETWORK_CONTROLLER:
            devclass = DEV_CLASS_NET;
            devcap = DEV_CAP_BUSMASTER;
            break;

        case MASS_STORAGE_CONTROLLER:
            devclass = DEV_CLASS_STORAGE;
            devcap = DEV_CAP_BUSMASTER;
            break;

        case BRIDGE_DEVICE:
            devclass = DEV_CLASS_BRIDGE;
            // A bridge device should be passive
            devcap = DEV_CAP_NOINT;
            // If one of the following tests fail, we fall through to the
            // default case
            // For now, we ignore all but pci-to-pci bridges
            // Include only if bridge has a secondary/subordinate bus
            // (0 is root)
            if (pci_probelist[i].class == PCIDF_CLASS_BRIDGE_DEVICE_PCI_PCI
                && pci_1_header->ph1_secondary_bus != 0
                || pci_1_header->ph1_subordinate_bus != 0)
                    break;

        default:
            devclass = DEV_CLASS_UNKNOWN;
            break;
        }
    }
}
```

If a device is found, its bus device function (BDF) code is returned and stored for future reference when the device's configuration space is read or written. The BDF is a unique identifier for devices connected to the PCI subsystem, and is composed of three components. The *bus number* is the index of the bus to which the device is attached. A single PCI subsystem supports a maximum of 256 buses. The *device number* is the slot number on a given bus. Although the theoretical maximum number of devices on a single bus is 32, the actual maximum is often less due to electrical limitations. The *function* is the addressed controller on the expansion card. A single device may implement up to eight functions, but at least one is required.[1]

The BDF is used for routing reads and writes to the configuration space of a specific device. The extended PCI configuration space contains information about device capabilities, such as power management, MSI/MSI-X, and PCIe. Also found here are the base address registers (BARs) that reveal the memory addresses used by the device controller. The Vortex PCI code implements an interface for reading and writing to the PCI configuration space, and setting device power state. This is part of the *device configuration interface*, and is made available through the device structure displayed in Code Snippet 3.3.

Code Snippet 3.3: The Vortex device structure contains all necessary information about a device. If a device is located on a secondary bus, *`de_parent` points to the bridge device that manages the bus, which can be used to configure pin-based interrupts if the targeted device is not present in the MP-table.

```c
struct device_t {
    device_t       *next, *prev;
    device_t       *de_parent;              // Parent device if located on a
                                            // secondary bus
    devclass_t     de_class;                // Device class
    devtype_t      de_type;                 // Device type
    devcap_t       de_cap;                  // Device capabilities
    enum_t         de_index;                // Device unit number
    devstate_t     de_state;                // Device state
    resourceid_t   de_resourceid_write;     // Resourceid for write
    resourceid_t   de_resourceid_read;      // Resourceid for read
    resourceid_t   de_resourceid_readwrite; // Resourceid for read/write
    resourceid_t   de_resourceid_interrupt; // resourceid for interrupts
    devclassdrv_t  *de_classdrv;            // Device class driver
    devconf_t      *de_conf;                // Device configuration
    devdrv_t       *de_drv;                 // Device driver
    devbuf_t       *de_read_done;           // Completed reads
    devbuf_t       *de_write_done;          // Completed writes
    devpowerdrv_t  *de_pwrdrv;              // Device power driver
};
```

1. An example of a multi-function device, is a card that includes both an audio controller and a Universal Serial Bus (USB) hub.

We add an optional reference to a parent bridge to the device structure, and extend the probe function to match devices with their parent bridges. The function displayed in Code Snippet 3.4 is invoked for every device in the global device list, and populates the parent pointers. To do so, we use the information present in the configuration space of a bridge device, which includes a number representing the ID of the bus to which the bridge expands.[2] Figure 3.4 depicts the detected PCI hierarchy of our server.[3]

Code Snippet 3.4: The pci_configure_parent function is run per device immediately after all PCI devices are located, and stores a reference to the parent bridge.

```
static void pci_configure_parent(device_t *newdev)
{
    device_t *dev;
    pcidev_t *parentpcidev, *newpcidev;
    pcidf_header1_t *bridgeheader;

    newpcidev = (pcidev_t *)newdev->de_conf->dc_priv;

    if (newpcidev->pd_busdevfunc == 0)
        return;

    for (dev = VxL_QHEAD(get_device_list()); dev != NULL; dev = dev->next)
    {
        parentpcidev = (pcidev_t *)dev->de_conf->dc_priv;

        // The new device cannot be its own parent
        // and non-bridge devices cannot be parents
        if (dev == newdev ||
            PCI_CLASSCODE(parentpcidev->pd_class) !=
            PCIDF_CLASSCODE_BRIDGE_DEVICE)
            continue;

        // Read the header as a PCI header type 1
        // which specifies secondary bus
        bridgeheader =
            (pcidf_header1_t *) &parentpcidev->pd_config.pc_basic.pc_header0;

        if (bridgeheader->ph1_secondary_bus == PCIDEV_BUS(newpcidev))
        {
            newdev->de_parent = dev;
            return;
        }
    }
}
```

2. While a non-bridge device implements PCI header type 0 that includes six BARs, bridges implement PCI header type 1, which replaces four of these registers with information about the expansion bus. This information includes the Secondary Bus, and Subordinate Bus Number, that identifies the bus controlled by the bridge [28].
3. The number of detected devices is actually much higher, and includes other kinds of bridges and devices, which we ignore in our system.

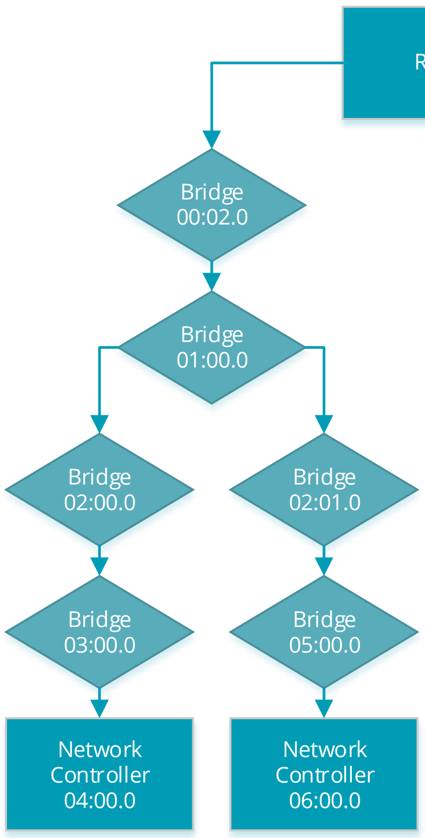


Figure 3.4: PCI hierarchy displaying the route to our network and storage controllers.

3.6 Determining IRQ Information

Pin-based interrupts use a dedicated wire to transmit the interrupt signal unlike MSI, where interrupts are transmitted with in-band signaling. A PCI device can have up to four pins [51], identified as INTA#, INTB#, INTC#, and INTD#. However, devices that implement a single function are limited to using only one pin.

When configuring pin-based interrupts for devices, Vortex performs a lookup in the MP table to get the interrupt pin and vector that the device is configured to use. If the information is not found, the device remains unconfigured. With the introduction of parent devices, we are able to detect the information based on the PCI hierarchy.

3.6.1 Interrupt Routing

In a PCI subsystem where devices are connected through expansion buses, the actual interrupt line used for each device may differ based on its index on the bus. For example, if a device specifies that its interrupt line is INTA#, but is the second device on the bus, the CPU will receive interrupts from the device via line INTD#. The PCI bridge specification defines a routing table for interrupt lines [28], this is shown in Table 3.1.

Table 3.1: APIC Interrupt routing table.

Input	Device 0	Device 1	Device 2	Device 3
INTA#	INTA#	INTD#	INTC#	INTB#
INTB#	INTB#	INTA#	INTD#	INTC#
INTC#	INTC#	INTB#	INTA#	INTD#
INTD#	INTD#	INTC#	INTB#	INTA#

3.6.2 Swizzling

Based on the interrupt routing table, we can use Equation 2 to calculate the output pin based on the input pin. If we have a device that is far down the PCI hierarchy, such as the network controllers in Figure 3.4, the calculation must be done for every bridge between the device and the root complex—a process nicknamed *swizzling*.[4]

$$\text{pin}_{\text{parent}} = (\text{bus}_{\text{child}} + \text{pin}_{\text{child}}) \bmod 4$$

Equation 2: PCI-to-PCI Bridge swizzle.

We implement the swizzling procedure in the `mpc_intdetail_init` function, which previously only performed a single lookup in the MP table for a single device. This function, shown in Code Snippet 3.5, now invokes the parent bridge of a device whose information was not found, and continues up the hierarchy until the root is reached, or if a pin matches the original pin, at which point there is no change in routing (see Table 3.1).

4. *Swizzling*, meaning stirring or mixing, is a widespread term in open source implementations of PCI bridge drivers, including the ones used in the FreeBSD and Linux Kernels, and describes the calculation of which interrupt line is used.

Code Snippet 3.5: The function used to read IRQ information from the MP table can also determine this by doing a swizzle through the PCI hierarchy.

```c
vxerr_t mpc_intdetail_init(intdetail_t *intdetail, device_t *dev)
{
    // Variables and initialization
            {···}
    // Get the initial pin from the device's configuration space
    pin = cfg->pc_basic.pc_header0.ph0_interrupt_pin;

    for (tmp_dev = dev; tmp_dev;)
    {
        // Locate interrupt information in MPC tables
        // mpc_interrupt_num is the a number of allocated entries in the table
        for (i = 0; i < mpc_interrupt_num; i++) {
            // Collect BUS and DEVICE id from current entry
                    {···}

            // Move to next entry if this is not the one we're looking for
            // (entry is not for this bus)
            if (source_bus_id != devid->pi_busid)
                continue;

            // If the MP table entry matches the BUS and DEVICE ID of the device,
            // copy the interrupt information from the MP table
            if (source_bus_id == devid->pi_busid &&
                source_bus_irq == devid->pi_devbusid) {
                // COPY interrupt information from MP table
                            {···}

                // If the entry is a direct match we are good to go, else, we
                // see if we can find a better match
                if (pin == source_bus_pin)
                    return VXERR_OK;
            }
        }
        // Get parent bridge of current device if it exists, or end here
        tmp_dev = tmp_dev->de_parent;
        if (tmp_dev == NULL)
            break;

        // Assume PCI type of device id
        devid = device_config_get_id(&nbytes, tmp_dev);

        // Determine type of bus
        bustype = mpc_dev_to_bustype(devid);
        if (bustype == -1) {
            vxerr = VXERR_NEXIST;
            goto error;
        }
        // Swizzle interrupts (bridge interrupt routing) (use pin id 0-3, not 1-4)
        pin = (((pin - 1) + devid->pi_devbusid) % 4);
    }
    return VXERR_OK;
error:
    return vxerr;
}
```

3.7 Summary

This chapter described improvements to the Vortex device system that allow configuration of devices that the BIOS is unable to recognize. We have implemented a system for detecting the hierarchy of the PCI subsystem, and described how this is used to determine IRQ information that is missing from the MP table. This solves a problem that occurred when we attached an NVMe device to the server, which hindered us from using it, and by adding this support, we are now able to detect and set up pin-based interrupts for our NVMe device.

4

NVMe as a SCSI Device

This chapter describes the implementation of an NVMe driver for Vortex that is exposed as a SCSI device, allowing the current storage stack to remain untouched. Configuration of NVMe controllers did necessitate changes to Vortex and these changes are presented here.

We open this chapter with a presentation of the current state of the Vortex storage stack and how we aim to integrate NVMe in it. Next, we detail how we leverage the improvements described in Chapter 3 to configure NVMe devices when the BIOS does not recognize them, and describe how we prepare our NVMe device for use. We then present the details of how we can expose our NVMe SSD as a SCSI device, and how normal operation occurs. Finally, we present our configurable methods for selecting the queue to which we submit a command, and end the chapter with a summary.

4.1 The Vortex Storage Stack

Vortex already contains a fully functional MegaRAID Firmware Interface (MFI) driver that provides support for the MegaRAID SAS family of RAID controllers. A detected MFI device is exported as SCSI capable, allowing it to be paired with the Vortex SCSI driver, which is registered as a SCSI resource. When a process issues a read or write request to physical storage, it is passed through a generic storage device resource to the SCSI resource, which translates the request to a

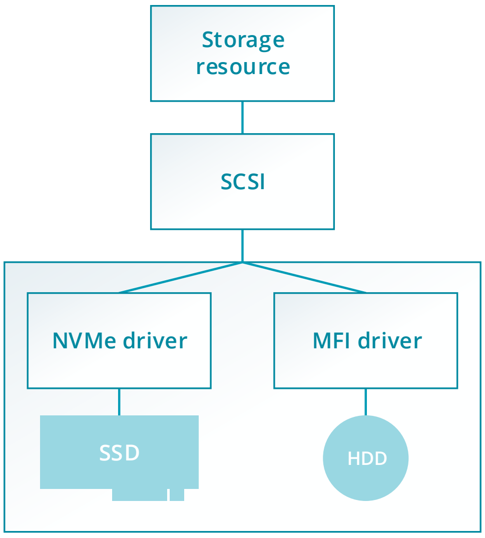


Figure 4.1: We report SCSI capabilities from our NVMe driver, and allow the current storage stack to remain unchanged, regardless of the underlying physical device and storage interface.

corresponding SCSI command that is sent to the MFI device driver.

To start using NVMe in Vortex, driver software is required. NVMe drivers are already shipped with Windows, FreeBSD, and many Linux based OSs. Vortex is not an extension of an existing OS, and cannot use drivers implemented for mainstream OSs. Some low-level work is involved when implementing a driver for devices such as NVMe. This includes designing a usable API for reading and writing to device registers, and setting up command structure and logic for safely issuing commands. A few of these changes were implemented previously as a capstone project [52], and are not covered here.

We aim to reuse as much as possible of the current storage stack, and expose the NVMe device as a SCSI device. Figure 4.1 shows the desired result, with NVMe placed in juxtaposition to MFI.

4.2 Controller Initialization

The NVMe specification states that initialization of the controller *should* be done via pin-based interrupts or single-message MSI. It is in other words optional to support configuration through MSI-X. Our Intel DC P3600 SSD does not have MSI capabilities, nor does it implement support for configuration over MSI-X. As described in Chapter 3, our DELL PowerEdge M600 blade server fails to recognize the drive, and two implications emerge. The first is that the OS is now responsible for both detection and configuration of the device in order to allow applications to use it. The second implication is that because the device is not recognized by the BIOS, the disk cannot be used as a boot device.

Vortex prioritizes MSI-X over MSI, and MSI over pin-based interrupts, and configuration of peripheral devices in Vortex works under the assumption that a device that supports multiple interrupt models, will become fully functional with any of them. For NVMe, this is not true, and we need to override the initial configuration to use pin-based interrupts.

All drivers in Vortex implement a standard set of functions—*a driver multiplexor*—that is part of the device interface, and is used to connect the driver to the rest of the resource system. Our NVMe driver implements the interface listed in Code Snippet 4.1, which is used by the virtual dispatch table to route requests.

Code Snippet 4.1: The NVMe driver multiplexor is a set of functions that is used by the virtual dispatch table to route a request to the correct device.

```
static devdrvmux_t nvme_drvmux = {
    .ddm_probe      = (ddm_probe_t)     nvme_probe,
    .ddm_get_cap    = (ddm_get_cap_t)   nvme_getcap,
    .ddm_start      = (ddm_start_t)     nvme_start,
    .ddm_stop       = (ddm_stop_t)      nvme_stop,
    .ddm_interrupt  = (ddm_interrupt_t) nvme_interrupt,
    .ddm_readwrite  = (ddm_readwrite_t) nvme_readwrite,
    .ddm_watchdog   = (ddm_watchdog_t)  nvme_watchdog,
    .ddm_get_id     = (ddm_get_id_t)    nvme_getid,
};
```

A ddm_probe_t function is used to match a driver with a device, and returns an instance of the controller object shown in Code Snippet 4.2. This structure is used as an argument when any of the other functions in the device multiplexor is called, to reference a specific device. The ddm_get_cap_t function is used to report capabilities of the driver. For example, the SCSI resource invokes this function for a device to determine whether it is SCSI capable. In the NVMe driver, we use this to our advantage, and add a new device capability flag, DEV_CAP_INTPIN_INIT, which is detected when the kernel attempts to configure interrupts.

During system initialization, device_config_set_inthandler is called for every discovered device, and allocates the number of interrupt vectors supported by the device.[1] We use the DEV_CAP_INTPIN_INIT flag when allocating inter-

1. The number of supported interrupts for an MSI/MSI-X compatible device is read out of the extended configuration space in its PCI header.

rupts for the device to determine whether it should ignore MSI-X and configure pin-based interrupts instead. If the capability is set, interrupts for the device are initialized based on information located in the MP table, or based on the swizzling process described in Chapter 3. When the NVMe device is configured and ready for use, the NVMe driver removes `DEV_CAP_INTPIN_INIT` from its capabilities, and issues a new call to `device_config_set_inthandler`, which enables MSI-X for the device.

Code Snippet 4.2: The `nvme_controller_t` structure contains all information necessary to interact with an NVMe device.

```
struct nvme_controller_t {
    // The actual device handled by this driver instance
    device_t                                    *dev;
    // Used to set/unset DEV_CAP_INTPIN_INIT
    bool_t                                      ctrlr_ready;
    // Registers that will be cached during initialization to
    // reduce unnecessary access
    nvme_register_identify_controller_data_t   *controller_id;
    nvme_register_controller_capabilities_t    cap;
    // Read from device
    // Defines the offset between doorbell registers
    // Typically 0 (4 byte), but may be different, especially
    // in software emulations
    uint16                                      doorbell_stride;
    // Number of interrupts allocated for device
    uint32                                      max_numint;
    // Number of allocated queue pairs
    // This is set during controller initialization
    uint32                                      num_queue_pairs;
    // List of namespaces on this device
    nvme_namespace_t                            *namespaces;
    // Only commands that are part of the Admin Command Set may be
    // issued to the Admin Submission Queue
    nvme_admin_queue_pair_t                     admin_queue_pair;
    // Admin commands cannot be issued to regular I/O queue pairs
    nvme_queue_pair_t                           *queue_pair;
};
```

Except for interrupt allocation and configuration of pin-based interrupts, the device is prepared for use from `nvme_start`. From here, we configure the desired command arbitration method, the submission and completion queue pairs, and enable the controller to make it ready to process I/O requests.

4.3 Setting up I/O Queues

The clear separation of administrative and I/O specific commands defined by NVMe is preserved in our driver. Although all commands are equal in size (64 bytes), the admin submission queues (ASQs) and IOQs only accept commands from the admin and NVM specific command sets, respectively, and we therefore implement a logical separation between the two.

An NVMe queue is a circular buffer allocated in DMA memory accessible by the controller. For a queue of size n, a submission queue (SQ) occupies at any time $n \times 64$ bytes, while a completion queue (CQ) occupies $n \times 16$ bytes. Our driver always creates queues in pairs, and each queue is referenced via an instance of the `nvme_queue_pair_t` struct. The controller object contains an array of all allocated queue pairs, including a standalone reference to the admin queue (AQ) pair. Along with the SQs and CQs, we associate a third circular buffer with each queue pair, in which information about requests that are currently being processed are stored. The `nvme_completion_status_t` structure, displayed in Code Snippet 4.3, includes a callback function to call when the command completes, as well as a reference to the original I/O request.

Code Snippet 4.3: A `nvme_completion_status_t` is associated with each issued command.

```
struct nvme_completion_status_t {
    // Called when command completes
    nvme_completion_callback_t  completion_callback;
    // Associated CQ entry, contains command specific status
    nvme_completion_t           *completion;
    // Associated command ID
    uint16                      cid;
    // Driver specific status, occupied or free
    nvme_command_status_code    command_status;
    // I/O request
    devbuf_t                    *dbuf;
    // PRP List for use with reads and writes over one page
    prp_entry_t                 prp_list[64];
};
```

4.3.1 Physical Region Pages

The host memory locations used for data transfers are specified using Physical Region Page (PRP) entries, and are used as a scatter/gather mechanism. The size of a page is set during controller initialization, and is in Vortex fixed to 4 KB. Each command includes two Physical Region Page (PRP) fields, and supports transfers of 8 KB. Additionally, the second PRP entry may point to a memory page that contains a list of PRP entries, as illustrated in Figure 4.2. The list contains up to 64 entries, although the controller may support less, or even more, in which case the last entry points to an additional PRP list. The number of entries used per command is implied by the command parameters and memory page size.

The structure in Code Snippet 4.3 includes a PRP list that can be used when

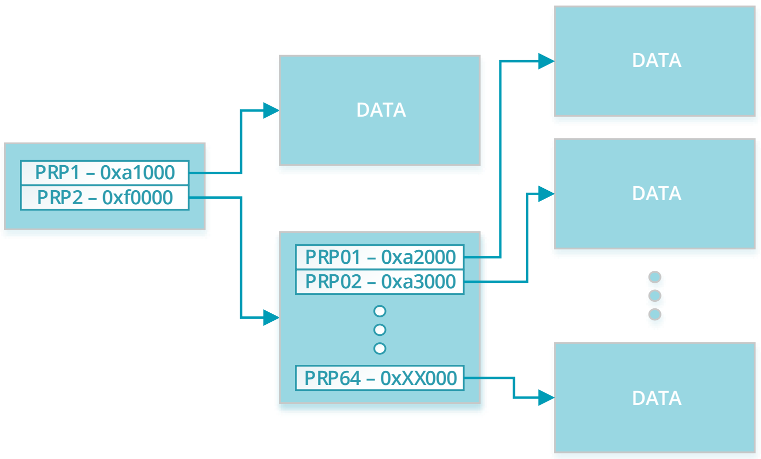


Figure 4.2: PRP entries point to a memory page used for the data transfer. If the data transfer is for more than two memory pages of data, the second PRP entry in the command may point to a list of PRP entries.

required by a request. We implement a one-to-one association between a command entry and a completion status entry to avoid race conditions, and do the allocation of all entries once, at the time of controller initialization to avoid frequent memory allocation/deallocation.

The entries in a PRP list are not required to be physically contiguous. However, the buffers allocated for data transfer in Vortex are always contiguous, making the process of creating a PRP list straightforward.

4.4 Exposing NVMe as a SCSI Device

In addition to using the `ddm_get_cap_t` function to change how interrupts are initially configured, we also use this to report that our NVMe device is SCSI capable. This is picked up by the SCSI resource, which starts to query the driver for more information.

SCSI uses *inquiry* messages to probe a storage device for information such as its type, supported command set, number of LUNs, and storage capacity. For a SCSI device, these commands are sent to, and interpreted by, the controller. Based on the translation reference [53] released in January 2015, we implement a translation between the SCSI commands received from the SCSI resource and the information gathered from the storage device using NVMe's *identify controller* and *identify namespace* commands.

4.4.1 Normal Driver Operation

All commands directed to the physical device go through the `nvme_readwrite` command shown in Code Snippet 4.4. Here, repeated calls are made to `device_devbuf_next` to retrieve *devbufs*, each detailing an I/O operation, such as a read or a write, the address of buffers, and the size of the request.

Code Snippet 4.4: During normal operation, `nvme_readwrite` is called for all commands directed to the storage device. We instruct the compiler with branch prediction information to favor read and write.

```c
static vxerr_t nvme_readwrite(nvme_controller_t *nvme)
{
        {···}
    // Get next request
    dbuf = device_devbuf_next();
    if (__builtin_expect((dbuf == NULL), 0) {
        vxerr = VXERR_OK;
        goto error;
    }
    ext = (dbufstorage_t *) dbuf->db_classext;
    // Respond immediately to deletes, but assume that it is not a delete
    if (__builtin_expect(ext->dse_state & SBUF_STATE_DELETE), 0) {
        assert(ext->dse_state & SBUF_STATE_CMD);
        vxerr = VXERR_OK;
        goto iodone;
    }
    // Assume that command is read or write
    // The SCSI resource only issues READ/WRITE 16 commands
    cmd = (uint8*) ext->dse_cmd;
    if (__builitin_expect(cmd[0] == SCSIDF_COMMAND_OPCODE_READ_16 ||
            cmd[0] == SCSIDF_COMMAND_OPCODE_WRITE_16))
    {
        if (cmd[0] == SCSIDF_COMMAND_OPCODE_READ_16)
            vxerr = nvme_cmd_read({···});
        else if (cmd[0] == SCSIDF_COMMAND_OPCODE_WRITE_16)
            vxerr = nvme_cmd_write({···});
    }
    else
    {
        switch (cmd[0])
        {
        case SCSIDF_COMMAND_OPCODE_INQUIRY:
                {···}
        case SCSIDF_COMMAND_OPCODE_REPORT_LUNS:
                {···}
        case SCSIDF_COMMAND_OPCODE_READ_CAPACITY_10:
                {···}
        default:
                CRITICAL("DEV 0x%X Received unknown SCSI command 0x%x",
                    device_get_index(nvme->dev), cmd[0]);
        }
    }
        {···}
}
```

Other commands, such as SCSI *inquiry* and *report LUNs*, also arrive in this function. These requests are issued rarely—only during SCSI resource initializing—and by using GCC's preprocessor flag `__builtin_expect`, we provide the compiler with branch prediction information. In most cases, programmers are bad at predicting how a program will execute, but in this case, we know that reads and writes will occur the most, and this is the path for which we want to optimize our function.

4.4.2 Command Completion

For admin commands, we implement both synchronous and asynchronous handling of completions. Asynchronous completion is automatically selected by specifying a callback to run on completion. Otherwise, the original code path is reentered only when the command completes. Synchronous operation is required during some stages in the initialization of the controller, as we need to ensure that a command succeeds before issuing the next. For example, when registering queues with the controller, it is important that a CQ is successfully created before we register the associated SQ. On the other hand, we only implement asynchronous completion for IOQs.

When the controller completes a command for an asynchronous request, it generates an interrupt, which arrives in the `nvme_interrupt` handler shown in Code Snippet 4.5. This handler receives all interrupts generated by the controller, and because we have a logical separation between the AQ and IOQs, we need to invoke two different handlers based on the incoming interrupt vector. The switch is, however, because vector 0 always refers to the AQ, while all other vectors refer to an IOQs. To ensure an optimized completion path for IOQ operation, we provide the compiler with branch prediction information here as well.

Code Snippet 4.5: The interrupt handler uses branch prediction to favor handling of I/O completions over admin completions.

```c
static vxerr_t nvme_interrupt(nvme_controller_t *nvme, uint32 devintvector)
{
    // We know that most interrupts are generated for I/O queues
    // and provide GCC with branch prediction information such that
    // processing of the admin queue becomes the 'abnormal behavior'
    if (__builtin_expect((devintvector != 0), 1))
        nvme_process_completion(nvme, devintvector);
    else
        nvme_process_admin_completion(nvme);

    return VXERR_OK;
}
```

All NVMe IOQs are given a unique identifier, but rather than assigning this identifier to a variable in the queue structure, we use the index of the queue pair in the array in which we store reference to the queue pairs. The vector assigned to each queue also reflects its identifier, and because we separate the AQ from the IOQs, the queue pair in question is located at index $qid - 1$.[2] When receiving a completion for an IOQ, the number of actually completed commands are not restricted to one. As shown in Code Snippet 4.6, we continue processing completions as long as there are any. This is possible because of the phase bit included in the completion entry, which the controller flips every time the queue wraps (the queue wraps when it reaches the end and returns to offset 0 in the queue).

Code Snippet 4.6: IOQ specific interrupt handler. All completions in the current phase are processed.

```c
void nvme_process_completion(nvme_controller_t *nvme, uint16 qid)
{
    nvme_completion_queue_t *cq;
    volatile nvme_completion_t *completion;
    nvme_completion_status_t *completion_status;
    cq = NVME_COMPLETION_QUEUE(nvme, qid);
    while (1)
    {
        completion = &cq->base[cq->head];
        // Continue processing all completions for this phase
        if (completion->status_field.p != cq->phase)
            break;

        // Get associated completion status
        completion_status = &NVME_COMPLETION_STATUS(nvme, qid, completion->cid);

        // Add reference to the completion status
        completion_status->completion = (nvme_completion_t *)completion;

        // Increment head, wrap if end of queue and switch phase
        if (++cq->head == cq->size)
        {
            cq->head = 0;
            cq->phase = !cq->phase;
        }
        // Controller has completed the command, but we are not done with it yet
        completion_status->command_status = NVME_COMMAND_STATUS_COMPLETE;
        completion_status->completion_callback(nvme, completion_status);

        // The work is done, signal controller that we are happy
        completion_status->command_status = NVME_COMMAND_STATUS_FREE;
        NVME_SUBMISSION_QUEUE(nvme, qid)->head = completion->sqhd;
        nvme_register_write_completion_queue_head_doorbell(nvme, qid, cq->head);
    }
}
```

2. Queue identifier and interrupt vector 0 always refers to the AQ.

4.5 Driver Specific Command Arbitration

Drivers for mainstream OSs, such as FreeBSD, implement one queue per core [54]. We want to explore how to best utilize the number of IOQs supported by NVMe. We implement in our driver a customizable method for selecting the I/O submission queue (IOSQ) we use to submit a command. This is a driver specific configuration, and does not affect the arbitration method, nor command burst settings, of the controller. The current implementation features three different driver specific arbitration methods: RR, per core, and per core pool.

4.5.1 Round Robin

The first and simplest implementation is a RR selection of queues. With this configuration we use all available queues. An Intel DC P3600 SSD supports 31 I/O queue pairs, each supporting 4096 commands. This means that by using this configuration, we could issue 126 945 commands before having to wait for completions.[3]

All queues can be used by any CPU, which means that locks must be applied. The current active queue is a shared variable, which is incremented when the queue it represents is successfully locked, to avoid race conditions. The queue selection mechanism is displayed in Code Snippet 4.7.

Code Snippet 4.7: The driver-specific RR arbitration method loops without concern for CPU affinity, and requires lock primitives to guard the active queues.

```
vxerr_t nvme_readwrite({···})
{
    lock_t  lock;
    static uint16 queue_id = 0;
    VxO_LOCKNULL(&lock);
    VxO_LOCK(&lock, NVME_SUBMISSION_QUEUE(nvme, queue_id)); // Lock queue
    // We only get here if we have the lock
    if (++queue_id > nvme->num_queue_pairs)
        queue_id = 1;

        {···} // calculate number of blocks in request and offset
        {···} // generate prp list if required
        {···} // issue read/write

    VxO_UNLOCK(&lock);
    return vxerr;
}
```

3. Although the maximum number of entries is 4096, the queue is considered full when the tail is one entry behind the head.

4.5.2 Per Core

The second implementation is an approach used by most mainstream driver implementations, including the FreeBSD NVMe driver, and is a configuration in which we allocate a single queue per available core in the system. An eight core system supports having 32 760 outstanding commands using this setting.

We show the simple logic used to select the active queue in Code Snippet 4.8. The buffers containing information about the I/O request also include the ID of the CPU that issued it. This ID is used to select the queue, which results in a fixed affinity, meaning that queues are not shared between cores, such that no locks are required.

Code Snippet 4.8: The driver-specific *per core* arbitration method uses one queue per CPU core. Queues have fixed affinity, and are not shared among cores, thus locks are not required.

```
vxerr_t nvme_readwrite({···})
{
    uint16 queue_id;
    // CPU ID’s start with 0, IOQ ID’s start with 1
    queue_id = CPUID + 1;

        {···} // calculate number of blocks in request and offset
        {···} // generate prp list if required
        {···} // issue read/write

    return vxerr;
}
```

4.5.3 Per Core Pool

Our third arbitration variant is a a combination of both the RR and *per core* methods, which we call a *per core pool* arbitration method. With this setting, we allocate an equal number of queues per available core, and do an RR selection on a per core basis. For our eight core server, and the 31 queues supported by the controller, we allocate 3 queues per core, or 24 in total.

This setting requires more logic than the *per core* arbitration method, but the same benefits apply: fixed affinity allows lockless use of the queues. With the additional queues, the number of outstanding commands supported at any time is 12 285 per core, or 98 280 for the entire system.

Code Snippet 4.9: The driver-specific *per core pool* arbitration method uses a set of queues per CPU core. The number of allocated queues are always a multiple of the number of cores. Queue sets have fixed affinity, and are not shared among cores, thus locks are not required.

```
vxerr_t nvme_readwrite({···})
{
    static uint16 queues_per_core = NVME_NUM_QUEUE_PAIRS/CPU_MAXNUM;
    static uint16 queue_pool[CPU_MAXNUM] = { 0 };
    uint16 affinity;
    uint16 core_queue_num;
    uint16 queue_id;

    affinity = dbuf->db_reqhdr.rh_target.affinity;
    core_queue_num = queue_pool[affinity];

    if (++queue_pool[affinity] > (queues_per_core-1))
        queue_pool[affinity] = 0;

    queue_id = ((affinity * queues_per_core) + core_queue_num) + 1;

        {···} // calculate number of blocks in request and offset
        {···} // generate prp list if required
        {···} // issue read/write

    return vxerr;
}
```

4.6 Summary

This chapter has presented the implementation of an NVMe driver that is exposed as a SCSI device. We have created a way of overriding the default selection of interrupt delivery system, and are able to successfully configure the NVMe controller. In addition, we have shown some design choices made to reduce overhead produced by frequent memory allocation/deallocation, and by exploiting the compiler's `__builtin_expect` to provide branch prediction information.

We have implemented three different methods for selecting the I/O queue to which we submit a command. These methods are evaluated in Chapter 5.

5 Evaluation

This chapter evaluates the integration of an NVMe SSD as a SCSI device in Vortex. We start by presenting the environment used during the experiments, before discussing the problems that occured when we first introduced an NVMe device to our system. We then describe our evaluation method, before presenting and discussing the achieved results.

5.1 Environment

The experiments described in this chapter are all executed on a DELL PowerEdge M600 blade server equipped with two Intel® Xeon® E5430 processors running at 2.66 GHz, 16 GB of DDR2-667 PC2-5300 fully buffered RAM with error-correcting code (ECC), and an NVMe v1.0 compliant Intel DC P3600 SSD connected to a PCIe 2.0 bus. The SSD supports 31 IOQs, each supporting up to 4096 command entries.

In experiments that target mechanical disks, two 10 K RPM SAS HDDs are used. These drives are connected through a MegaRAID SAS 1078 controller, and are configured in RAID 0, with stripe size of 1 MB, and with adaptive read ahead, a good basis for high throughput.

5.2 Controller Configuration

During the implementation of our NVMe driver, we discovered that NVMe cannot be initialized with MSI-X, and we were forced to implement a method of retrieving missing interrupt information. With this capability, we are able to set up pin-based interrupts for the unrecognized controller, and configure it. We are, however, not interested in using pin-based interrupts for normal I/O operations, and have improved the device system to support a switch between interrupt delivery methods.

In the early stages of NVMe initialization, it is important that each step is completed successfully before the next is issued. We therefore configure the device to use pin-based interrupts, as required, but having the kernel ignore them, and rather use a polling mechanism to do a synchronized wait for completion. However, this is not completely safe, as the code will keep spinning if a completion never arrives, but we have yet to experience this problem.

5.3 Measuring Performance

Most raw storage devices are accessed through a file system (FS) abstraction, even when used as dedicated storage for database management systems (DBMSs). FSs often employ caching [55] or other latency hiding techniques, such as heuristic-based approaches to prefetching data [56], that significantly reduce latency. On the other hand, a FS may also introduce negative overhead by using journaling and other metadata [57], [58]. When a system is to be used for applications that rely heavily on disk storage, for example DBMSs, both positive and negative effects of using a FS must be taken into consideration.

This thesis is concerned with the capabilities of the storage interface and medium. Therefore, we design our experiments to bypass the FS, such that raw throughput is measured, without the effects of caching, latency hiding, or having to handle journaling.

5.3.1 Collecting Data

The OKA uses schedulers interpositioned on communication paths to control resource-consumption of processes [10]. Instrumentation code in the kernel-side schedulers and inside the resource itself is used to report resource-consumption. To collect data during experiments, we use the Vortex Monitor [59], a tool that exploits the instrumentation code to extract and export performance data over the network.

The Vortex Monitor samples performance data at one second intervals. Some metrics are computed by comparing the delta between consecutive samples. This leads to some inaccuracy on startup, so we generally ignore the first few seconds of our experiments.

5.3.2 Diskhog

Diskhog is an application designed to generate a continuous read or write load towards a block-level storage device. Because we are able to extract performance data directly from the kernel, we do not implement measurement code in this application; it is only concerned with generating load.

When targeting the storage device on a block level, all reads and writes must be performed in block-multiples. To simulate a multi-tenant workload towards the storage device, diskhog can be configured to start multiple threads. Each Diskhog thread opens its own connection to the storage device for reading and writing, and is given its own private LBA range to isolate reads and writes

The scheduler implemented in the kernel automatically distributes threads over the available cores in the system, which means that our single process can have up to eight threads running without having competition for CPU-time.

5.4 Configuring an SSD with the Limitations of MFI

MegaRAID SAS controllers have an advantage over SATA with a queue depth of 128 or 256 commands versus 32. Here we compare the achieved throughput when configuring NVMe with the limitations of our SAS HDDs to demonstrate the power of SSDs without the additional perks of having multiple queues and greater queue depths.

The NVMe driver is configured to use one I/O queue pair with room for 128 commands, matching the configuration of our MFI controller. Figure 5.1 shows that the SSD achieves almost three times the throughput of the HDD.

An interesting observation here is that the achieved write throughput is much higher than for reads. When a request is issued, data is copied from the source to a request buffer owned by the kernel, from which it is written to the

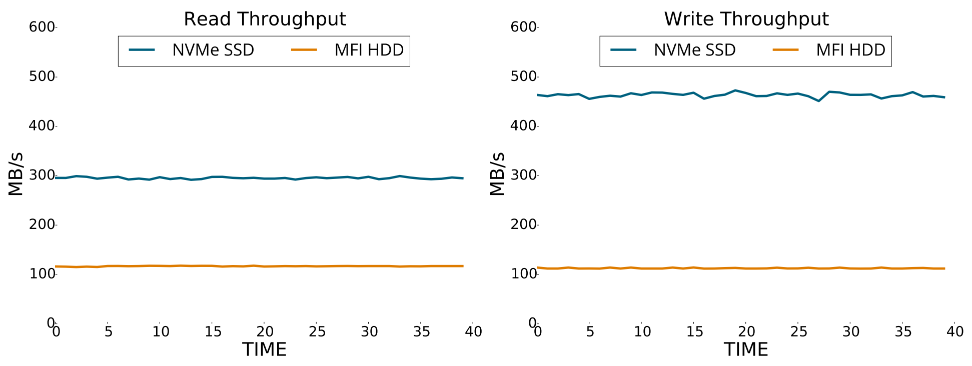


Figure 5.1: SSD vs HDD: Read and write throughput when the SSD is limited with an MFI configuration. One core producing load.

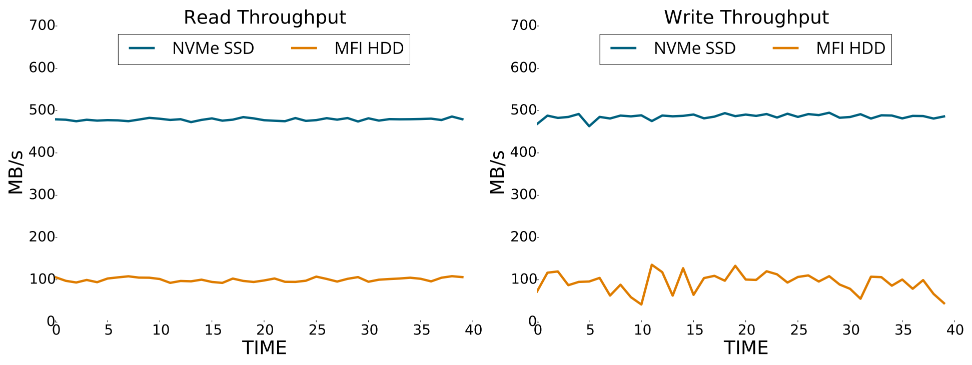


Figure 5.2: SSD vs HDD: Read and write throughput when the SSD is limited with an MFI configuration. Eight cores producing load.

destination. A process issuing a read must wait for the data to arrive in its local buffer, while a process issuing a write only has to wait for the kernel to copy the data into a request buffer, which explains why we are able to achieve higher write throughput.

Multi-Core Workload

Further, we simulate the workload of having multiple processes read from the storage device in parallel, by issuing requests from all eight cores simultaneously. As can be seen in Figure 5.2, the aggregate throughput generated from the SSD is almost five times the throughput of the HDD. Also visible here is that the maximum throughput does, however, not scale linearly with the number of cores. Subsequent experiments reveal the same results as shown in these graphs, and we choose to display a representable graph, rather than a "busy" one.

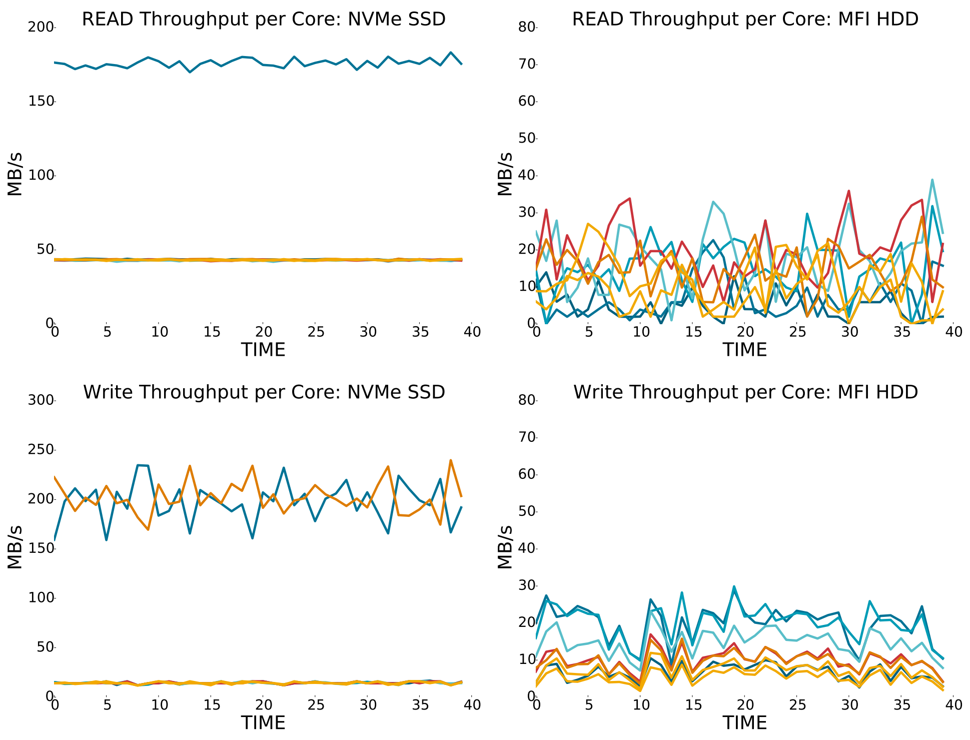


Figure 5.3: SSD vs HDD: Maximum achieved throughput per core when all eight cores produce load.

When inspecting the results displayed in Figure 5.3, which details the performance of each device individually, we see that achieved throughput per core is less for both devices, because all threads issuing requests share the same resource. We also observe that with our SSD, one core is achieving higher throughput than the other seven. This is because we use one global queue in this experiment, and guard it with a primitive spin-lock, which means that a core gets the lock completely by chance, and that one of the cores receive the lock more frequently than the rest. After multiple experiments, we found that the pattern is consistent, with only a difference in which core is favored.

5.5 Multiple Queues

We have seen that an NVMe SSD is better at handling parallel disk requests, even when limited with a single IOQ and a queue size of 128. When we configure our driver to use more than one queue, we see that lock contention is no longer a problem. Figure 5.4 shows that even when all queues are shared among all cores in the system, load distribution is even. Further, we find that the

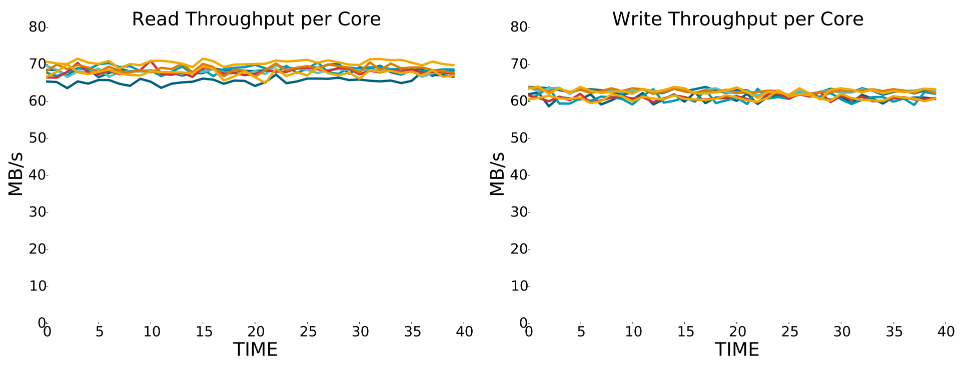


Figure 5.4: Throughput per core when multiple queues are used with the RR arbitration policy, and eight cores produce load.

maximum achieved throughput is achieved when only three cores produce load, as can be seen in Figure 5.5. The results are consistent across experiments here as well, and have negligible variance, making the graph representable.

Round robin

With the RR policy, all queues are shared between all cores in the system, and are prone to race conditions, thus requiring us to lock access to the device. We do not, however, need to lock access to the entire controller, but only to the current active queue.

When using the RR policy, we see that the achieved throughput is not much higher than with the limit of one queue. However, Figure 5.4 shows that because multiple queues are used, there is not as much contention, which results in an even load distribution.

Per Core

The per core policy matches that of drivers in mainstream OSs, with one queue allocated per CPU core in the system. The fixed affinity of each queue means that the locking mechanisms required in the RR policy are not needed.

Per Core Pool

Unlike other OSs, we also implement a variant of this, in which multiple queues are allocated per core and given fixed affinity. This allows us to support more

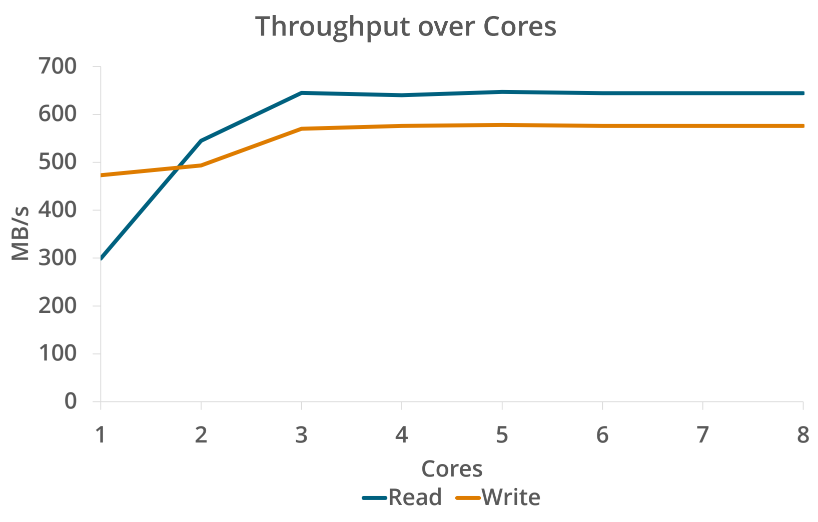


Figure 5.5: The maximum aggregate throughput as a function of cores producing load.

commands, and still require no locks.

5.5.1 Sustained Throughput

We measured read and write throughput for all implemented policies, and find that there is a tradeoff to making all queues available to all CPU cores. We also found that the achieved throughput is equal for the two policies that are based on fixed affinity, and that even though there is no better policy of the two, there is no backside to using more queues. On the contrary, if more queues are in use, then we are also able to handle more requests at a time.

Figure 5.6 shows a comparison between the different policies when performing I/O requests of size 4 MB, averaging the throughput over one minute. We have included the read and write throughput as measured with the *gnome-disks* benchmarking tool on an Ubuntu 14.04 (kernel version 3.13.0-24) server edition running on the same machine we use during our experiments in Vortex. We see that Ubuntu achieves 11 % faster reads and writes than Vortex. However, we observe that when we increase the request size, Ubuntu experiences a drop to about 560 MB/s, while the throughput achieved from Vortex remains unchanged. These results are consistent across runs, with small variance, so we have omitted error bars.

We believe that we should be able, after some optimization work, to match the throughput of Ubuntu, because we experience short bursts of higher throughput, up to about 720 MB/s just before our test application exits. When the process exits, it no longer produces I/O requests, but waits for the completion of all that remain, and it is plausible that because the CPU is free to only handle completion of commands, the achieved throughput is able to climb.

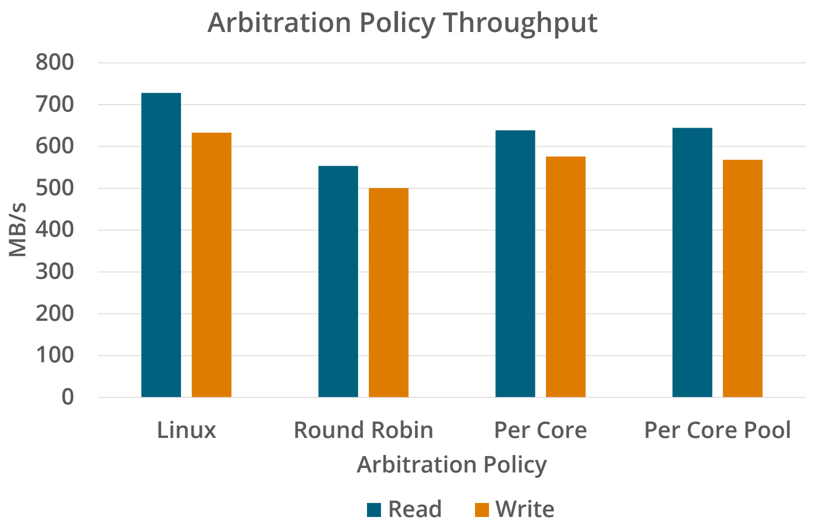


Figure 5.6: Sustained read and write throughput with different arbitration policies, including a comparison with Ubuntu.

5.5.2 I/O Operations per Second

In this experiment, we benchmark our driver to find the maximum number of IOPS that we are able to produce in our system. The experiment is performed by issuing 1000 4 KB requests per thread before waiting for results. As can be seen from Figure 5.7, when performing reads, we are able to complete close to 50 000 IOPS. For reads, the theoretical maximum for our disk is 70 000 IOPS, and we see that the results from testing throughput are mirrored here.

Note, however, that the number of IOPS for writes is much higher than the theoretical maximum for this disk, which is 30 000 IOPS. As explained in Section 5.4, writes are copied to kernel-side buffers for writes. Because we support a very large number of outstanding requests, the results just reflect the rate at which write requests are enqueued. We would prefer to measure

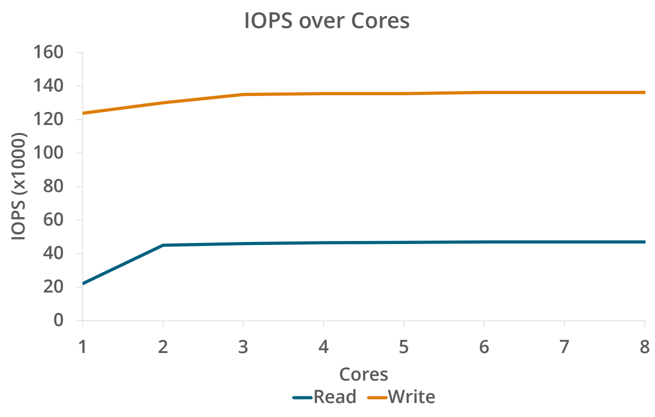


Figure 5.7: IOPS for random reads and writes of 4 KB data.

the rate at which requests are processed by the SSD. Unfortunately, the Vortex Monitor does not directly report this number, and time did not permit any workarounds for this issue.

When we compare Figure 5.5 to Figure 5.7, we see that only two cores are required to achieve maximum IOPS, while maximum sequential throughput is achieved with three cores. When generating load for maximum throughput, we issue and wait for a single large request to complete before issuing the next, while the IOPS experiment issues a large number of requests before waiting for replies. This reduces overall wait time and leads to better CPU utilization.

6 Concluding Remarks

In this chapter we summarize our contributions, and present deliberations on shortcomings and opportunities for future improvements to Vortex in the context of NVMe support.

6.1 Main Contributions

Conventional storage interfaces, such as SATA and SAS, are designed to interface with mechanical HDDs. However, SSDs are becoming more popular, and we see that the limitation in throughput no longer lies with the storage medium, but with the interface connecting the host and the disk. A solution to this has been to bypass the storage interface adapters, and connect the SSD directly to the PCIe bus. The NVMe standard is designed for PCIe-based SSDs, and offers high levels of parallelism and other features that are desirable for multi-tenant platforms.

NVMe devices require that initialization is done through pin-based interrupts, and the BIOS in our DELL PowerEdge M600 does not recognize our Intel DC P3600 SSD. We have implemented a PCI-to-PCI bridge device driver that is used to map the hierarchical structure of the PCI subsystem. We use this functionality to improve the Vortex system with the capability to determine IRQ information for devices that do not appear in the MP table. This functionality allows us to determine missing IRQ information and configure the SSD.

We have implemented an NVMe driver that exposes the NVMe SSD as a SCSI storage device, which allows us to use the SSD without introducing further changes to the Vortex storage stack. Our driver implements three command submission policies, which we have evaluated. Although our implementation has had little focus on optimization, we achieve higher throughput than what is attainable through a conventional SATA interface.

6.2 Future Work

Although functional, the integration of NVMe in Vortex as a SCSI device is not an optimal solution. The SCSI resource translates read and write requests to their corresponding SCSI commands and passes the request on to the device driver. The NVMe specific commands for read and write are standardized and do not differ between devices. This means that we can implement a resource interface for NVMe as part of the device driver, reducing the number of indirections between the calling process and the physical storage device.

As shown in Chapter 5, our solution does not match the throughput of the implementation of NVMe in a Linux based system, indicating that optimizations of our implementation are needed. One possible point of improvement is to increase the size of the kernel side request buffers, each of which can contain 32 KB of data. Our NVMe device supports transfers of up to 128 KB per command using PRP lists, but are currently limited by the request buffers.

6.2.1 Weighted Round Robin

The possibility to assign priority to processes through a scheduling mechanism in the storage controller is a completely new way of thinking, and is definitely worth exploring. However, Vortex implements schedulers for fair resource allocation on a high level, and it should be a point of focus to explore to what degree the controller should handle scheduling of queues.

This being said, the WRR arbitration funtionality also offers more control by management code. While RR treats all queues, including the admin queue, equally, WRR will always prioritize admin commands over any other I/O command. Additionally, the urgent priority class is always prioritized over queues belonging to the WRR priority class, and can be assigned to a queue controlled solely by the kernel if needed.

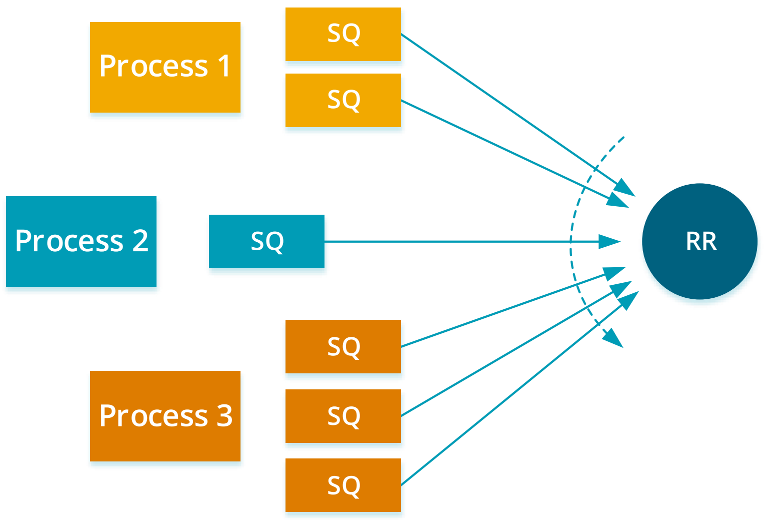


Figure 6.1: Given the support for enough I/O queues by the controller, the assignment of more queues to higher priority processes can be one way of assigning priority.

6.2.2 Alternative Methods of Assigning Priority

If WRR is not supported, we can still achieve priority-based command arbitration in other ways. Figure 6.1 illustrates a suggested solution, in which a higher prioritized process is given control over more IOQs.

Dataset Management

Optional features, such as the dataset management feature can be used to indicate attributes for LBA ranges. The controller can be instructed to optimize performance and reliability for ranges that are used for frequently accessed data, such as inodes for a file system, or swap area for the OS. This feature can possibly also be used to favor processes with higher priority.

6.2.3 Namespaces

The creation/deletion of namespaces is not trivial, and was, until version 1.2 of the NVM specification, only part of vendor specific command sets. When devices that do support this feature arrive on the market, we can use namespaces to support our isolation model, and allow multiple processes and VMs to access their own private namespace.

For example, during initialization of an NVMe device, a predetermined number of namespaces can be created, with equal shares of the available storage space, each of which can be assigned to a process or a VM. Alternatively, a namespace can be created along with a VM if it requires disk access.

6.2.4 Power Management

An NVMe device supports between one and 32 power states that can be managed statically or dynamically. Each power state defines the maximum power that the controller can consume, and the latency associated with each state. Static power management involves an active host-to-controller interaction for setting power state, while dynamic is handled by the controller by measuring the current load.

Vortex includes power management (PM) functionality that leverages information about resource consumption collected from kernel-side schedulers to make a decision about which power state the CPU should operate in, or whether it should enter an idle state [60]. This functionality can be extended to control the power state of NVMe devices as well.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits". *Electronics Magazine*, pp. 114–117, 1965.

[2] P. Rex Farrance, "Timeline: 50 Years of Hard Drives". [Online], Available: `http://www.pcworld.com/article/127105/article.html`. Retrieved: Mar. 17th, 2015.

[3] L. Mearian, "WD leapfrogs Seagate with world's highest capacity 10TB helium drive, new flash drives". [Online], Available: `http://www.computerworld.com/article/2604311/wd-leapfrogs-seagate-with-world-s-highest-capacity-10tb-helium-drive-new-flash-drives.html`. Retrieved: Mar. 23th, 2015.

[4] Seagate, "Seagate® Archive HDD ST8000AS0012". [Product Manual], Rev. A, July 2014.

[5] "ASTC Technology Roadmap". [Online], Available: `http://www.idema.org/wp-content/plugins/download-monitor/download.php?id=2244`. Retrieved: Mar. 23th, 2015.

[6] Toms Hardware, "Performance Charts Hard Drives". [Online], Available: `http://www.tomshardware.com/charts/hard-drives,3.html`. Retrieved: Mar. 17th, 2015.

[7] "Hitachi Deskstar 7K1000". Hitachi, [Data Sheet], 2007.

[8] "Micron P320h HHHL PCIe NAND SSD". Micron Technology Inc., [Datasheet], Rev. V 8/2014 EN, 2014.

[9] "NVM Express". NVM Express Inc, [Specification], Rev. 1.0e, Jan. 2013.

[10] Å. A. Kvalnes, D. Johansen, R. van Renesse, S. Valvag, and F. Schneider, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling". *IEEE Transactions on Parallel and Distributed Systems*,

Oct. 2014.

[11] Å. A. Kvalnes, "The Omni-Kernel Architecture: Scheduler Control Over All Resource Consumption in Multi-Core Computing Systems". Ph.D. dissertation, UiT The Arctic University of Norway, Oct. 2014.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization". *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor", in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[14] "Apache HTTP Server Project". [Online], Available: `http://httpd.apache.org/`. Retrieved: Mar. 29th, 2015.

[15] "MySQL:: The world's most popular open source database". [Online], Available: `http://www.mysql.com/`. Retrieved: Mar. 29th, 2015.

[16] "Apache Hadoop". [Online], Available: `https://hadoop.apache.org/`. Retrieved: Mar. 29th, 2015.

[17] A. Nordal, A. Kvalnes, J. Hurley, and D. Johansen, "Balava: Federating private and public clouds", in *Services (SERVICES), 2011 IEEE World Congress on*, IEEE, pp. 569–577, 2011.

[18] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing tcp", in *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, ACM, pp. 3–10, 2012.

[19] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young, and P. J. Denning, "Computing as a discipline". *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[20] L. A. Steen, "The science of patterns". *Science*, vol. 240, no. 29, p. 616, 1988.

[21] S. H. Hall, G. W. Hall, and J. A. McCall, "High-speed digital system design: a handbook of interconnect theory and design practices", Citeseer, 2000.

[22] "PCI-104 Specification". PC/104 Embedded Consortium, [Specification], Rev. v1.0, 2003.

[23] D. Giancoli, “Physics-principles with applications”, Aubrey Durkin, 2005.

[24] J. A. DeFalco, “Reflection and crosstalk in logic circuit interconnections”. *Spectrum, IEEE*, vol. 7, no. 7, pp. 44–50, 1970.

[25] “PCI-X 2.0: The Next Generation of Backward-Compatible PCI”. PCI-SIG, [White Paper], Rev. 1.0, 2002.

[26] M. Hachman, “Intel Begins Making Its Case Against PCI-X”. [Online], Available: http://www.extremetech.com/extreme/53584-intel-begins-making-its-case-against-pcix. Retrieved: May 12th, 2015.

[27] PCI-SIG, “I/O Virtualization”. [Online], Available: https://www.pcisig.com/specifications/iov/. Retrieved: May 12th, 2015.

[28] D. Anderson, T. Shanley, and R. Budruk, “PCI express system architecture”, Addison-Wesley Professional, 2004.

[29] “PCI Express Base Specification”. PCI-SIG, [Specification], Rev. 3.0, Nov. 2010.

[30] G. Field and P. M. Ridge, “The book of SCSI: I/O for the new millenium; 2nd ed.”, No Starch Press, 2000.

[31] T. P. Guide, “SCSI Bus Termination”. [Online], Available: http://www.pcguide.com/ref/hdd/if/scsi/cablesTermination-c.html. Retrieved: May 7th, 2015.

[32] C. M. Kozierok, “SFF-8020 / ATA Packet Interface (ATAPI)”. [Online], Available: http://www.pcguide.com/ref/hdd/if/ide/stdATAPI-c.html. Retrieved: May. 15th, 2015.

[33] A. Osborne, “An Introduction to Microcomputers: Basic concepts”, McGraw-Hill, 1980.

[34] “Faster Just Got Faster: SATA 6Gb/s”. [Online], Available: https://www.sata-io.org/system/files/member-downloads/SATA-6Gbs-Fast-Just-Got-Faster_2.pdf. Retrieved: May 7th, 2015.

[35] “Serial ATA AHCI”. Intel Corporation, [Specification], Rev. 1.3.1, 2014.

[36] J. Dedek, “Basics of SCSI”, Ancot Corporation, 1998.

[37] “Ultrastar SSD1000MR Enterprise Solid State Drives”. [Online],

Available: https://www.hgst.com/solid-state-storage/enterprise-ssd/sas-ssd/ultrastar-ssd1000mr. Retrieved: May 7th, 2015.

[38] "Seagate ST600MP0005". Seagate, [Data Sheet], 2014.

[39] "600 GByte, 3.5", 15,000 rpm Disk Drive Specification". Sun Microsystems, [Specification], no. 820-7290-10, Rev. A, Sept. 2009.

[40] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage". *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.

[41] "NAND Flash Translation Layer (NFTL)". Micron, [User Guide], 2011.

[42] "Intel Solid-State Drive 750 Series". Intel Corporation, [Product Specification], Apr. 2015.

[43] NVM Express Inc., "Why NVM Express". [Online], Available: http://nvmexpress.org/about/why-nvm-express/. Retrieved: May 11th, 2015.

[44] "Serial ATA". Serial ATA International Organization, [Specification], Rev. 3.2 Gold, Aug. 2013.

[45] "Connector Mating Matrix". [Online], Available: https://www.sata-io.org/sites/default/files/documents/MM_Nereus_Signage_Print_0719.pdf. Retrieved: May 19th, 2015.

[46] B. Gibbs, "SCSI Express - Advancements in PCIe Storage". [Online], Available: http://www.scsita.org/serial-storage-wire/2012/08/scsi-express-advancements-in-pcie-storage.html. Retrieved: May 19th, 2015.

[47] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable Multi-core Packet Processing in User-space", in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12, pp. 27–38, 2012.

[48] T. Shanley, "Plug and play system architecture", Addison-Wesley Professional, 1995.

[49] "PCI-to-PCI Bridge Architecture Specification". PCI-SIG, [Specification], Rev. 1.2, 2003.

[50] "MultiProcessor Specification". Intel Corporation, [Specification], 1997.

[51] D. Anderson and T. Shanley, “PCI system architecture”, Addison-Wesley Professional, 1999.

[52] K. Elsebø, “Vortex NVMe”. [Capstone Project], 2014.

[53] “NVM Express – SCSI Translation Reference”. NVM Express Inc, [Specification], Rev. 1.4, Jan. 2015.

[54] “FreeBSD NVMe Driver”. [Online], Available: `https://github.com/freebsd/freebsd/tree/master/sys/dev/nvme`. Retrieved: May 20th, 2015.

[55] “File Caching”. [Online], Available: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa364218%28v=vs.85%29.aspx`. Retrieved: May 16th, 2015.

[56] J. Griffioen and R. Appleton, “Reducing File System Latency using a Predictive Approach.”, in *USENIX Summer*, pp. 197–207, 1994.

[57] “Solaris™ ZFG and RED HAT Enterprise Linux EXT3 File System Performance”. Sun microsystems, Inc., [White Paper], June 2007.

[58] “File System Performance: The Solaris™ OS, UFS, Linux ext3, and ReiserFS”. Sun microsystems, Inc., [White Paper], Aug. 2004.

[59] R. M. Pettersen, “Hubble: a platform for developing apps that manage cloud applications and analyze their performance”. Master’s thesis, The University of Tromsø, 2012.

[60] J.-O. Karlberg, “ROPE: Reducing the Omni-kernel Power Expenses”. 2014.