PCI Express Multi-Root Switch Reconfiguration During System Operation

by

Heymian Wong

S.B. EECS, M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May, 2011
[June 2011]

Author: _____
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by: _____
Samuel R. Madden
Associate Professor of Electrical Engineering and Computer Science
M.I.T Thesis Supervisor
May 20, 2011

Certified by: _____
Johnny Chan
Senior Software Engineer, Network Appliance Inc.
VI-A Company Thesis Supervisor
May 20, 2011

Accepted by: _____
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

PCI Express Multi-Root Switch Reconfiguration During System Operation

by

Heymian Wong

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2011

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

Abstract

As the use of data continues to expand, reliance on enterprise datacenters have become more and more critical. Evaluation of modern day datacenter performance has showed that datacenter servers are severely underutilized, resulting in inefficient use of physical and software resources. In order to alleviate these concerns, datacenters must improve their resiliency, flexibility and scalability. CPU and memory virtualization are approaches that are becoming increasingly prevalent, but a major bottleneck still exists in I/O connectivity due to the physical constraints of I/O devices in a clustered system. One solution lies with the recently developed PCI Express multi-root partitionable switch. The partitionable switch is highly configurable and provides a means of altering the allocation of I/O devices amongst multiple hosts. This offers a range of unexplored possibilities to improve system flexibility. In the thesis, we investigate the feasibility of integrating and reconfiguring a PCI Express multi-root switch in a multi-host environment on a FreeBSD platform in a live system. We study modifications required of the PCI bus and system resources. We also detail the development of a device driver to handle dynamic reconfiguration of the switch and the system's PCI bus data structures.

MIT Thesis Supervisor: Samuel R. Madden
Title: Associate Professor of Electrical Engineering and Computer Science

VI-A Company Thesis Supervisor: Johnny Chan
Title: Senior Software Engineer, Network Appliance Inc.

Acknowledgements

This thesis would not be possible without all the support I received from MIT and Network Appliance (NetApp), my VI-A company. I express my sincere appreciation to Johnny Chan, my company advisor at NetApp for his constructive criticism, technical corrections and guidance in areas I found difficult. I must also thank all those that helped mentor me at NetApp, including my two highly respected managers, Steve Miller and Peter Snyder, and fellow colleagues in the Platform Hardware and Software groups for their perceptive comments, helpful suggestions and fruitful discussions.

I would also like to express my immense gratitude to my MIT advisor, Professor Sam Madden for generously taking me on as his advisee and for his supervision and encouragement over the course of my thesis research and writing process. And finally, I would like to thank my family for always being there for me, fully supporting me every step of the way. I would not be where I am today, if it weren't them all.

Table of Contents

1. Introduction **11**

1.1. Motivation 11

1.2. Objective and Scope 13

2. Background **15**

2.1. PCI Express 16

2.2. Multi-Root System Solutions 17

2.2.1. Non-Transparent Bridging (NTB) 17

2.2.2. I/O Virtualization 18

2.2.3. PCI Express Multi-Root Switch 19

2.3. PCI Express Switch 20

2.3.1. Standard PCI Express Switch 20

2.3.2. PCI Express Multi-Root Switch 20

2.4. PCI Hot-plug 22

3. PCI Express Operation and Specifications **24**

3.1. System Configuration - PCI Bus 24

3.2. Configuration Space Registers (CSR) 26

3.3. Transaction Routing 26

3.4. PCI Bus Enumeration 28

4. Prototype Hardware Configuration **30**

4.1. Host Systems 30

4.2. MR Switch Setup 30

4.2.1. PLX Setup 30

4.2.2. IDT Setup 31

4.2.3. Multi-Root Switch Selection 32

4.3. Endpoint devices 33

4.3.1. Intel Ethernet Card 33

4.3.2. NetApp NVRAM8 Card 33

4.4. Miscellaneous devices 34

4.4.1. Aardvark I2C connection – EEPROM access 34

4.4.2. VMETRO Network Bus Analyzer 34

5. Software **36**

5.1. PCI Bus Space Allocation and Enumeration 36

5.1.1. Method 1: Firmware Configurations 36

5.1.2. Method 2: FreeBSD Kernel Configurations 37

5.1.3. Method 3: Device Driver Configurations 40

5.2. MR Switch Device Driver Development 42
5.2.1. Device configurations 42
5.2.2. System Configurations 43
5.2.3. Device Driver Operations 44
5.2.4. Resource Allocation – Add Device 48
6. Experiment I: Re-configuration using Device Driver 52
6.1. Re-configuration with no I/O endpoints 52
6.2. Re-configuration with Ethernet Card 54
7. Experiment II: MR Switch Performance Validation 59
8. Discussion 61
8.1. Re-configuration Process 61
8.2. Performance 62
9. Conclusion 64
10. Bibliography 66

Table of Figures

Figure 2-1: A typical Northbridge-Southbridge layout. 15
Figure 2-2: Multi-root switch partitioning. 21
Figure 3-1: PCI bus topology 25
Figure 3-2: PCI Configuration Space for Type 1 and 0 Headers. 27
Figure 3-3: Virtual hierarchy of PCIe switch. 28
Figure 4-1: PLX MR Switch setup with two hosts. 31
Figure 4-2: IDT MR Switch setup with two hosts. 32
Figure 4-3: NetApp's NVRAM8 card. 33
Figure 4-4: Aardvark I2C/SPI Adapter setup for MR Switch EEPROM access. 34
Figure 4-5: VMetro's PCI Bus Analyzer 35
Figure 5-1: Pseudocode of modifications made to pci_pci.c. 38
Figure 5-2: Original PCI bus hierarchy. 39
Figure 5-3: PCI bus hierarchy with modified enumeration algorithm. 39
Figure 5-4: Functions for register access a device's CSR. 44
Figure 5-5: Functions for register access to a device's MMIO space. 45
Figure 5-6: Pseudocode to add a port to the MR switch. 45
Figure 5-7: Pseudocode to remove a port from the MR switch. 46
Figure 5-8: Pseudocode to resume a port from the MR Switch. 47
Figure 5-9: Pseudocode to suspend a port from the MR Switch. 47
Figure 5-10: Data structure storing range of allocated addresses. 49
Figure 5-11: Pseudocode for resource scan and allocation. 50
Figure 6-1: PCI bus from test with no I/O endpoints. 53
Figure 6-2: Updated PLX configuration register values. 54
Figure 6-3: Host 1 PCI bus after Ethernet card migration from Host 2. 55
Figure 6-4: The memory stack. 56
Figure 6-5: The I/O stack. 56
Figure 6-6: Updated registers in the switch configuration space. 57
Figure 6-7: Updated registers in the Ethernet device configuration space. 57
Figure 6-8: Receiving bandwidth of three Ethernet connections. 58
Figure 6-9: Transmitting bandwidth of three Ethernet connections. 58
Figure 7-1: Setup to measure latency through host system. 59
Figure 7-2: Setup to measure latency through the MR Switch. 59
Figure 7-3: Graphs the measured latencies from each test. 60
Figure 8-1: Graphs the increase in latency from the MR Switch. 63

1. Introduction

1.1. Motivation

As the use of data continues to expand, reliance on enterprise datacenters has become more and more common. Evaluation of modern day datacenter performance has shown that datacenter servers are severely underutilized. Furthermore, various concerns, such as environmental impact, company budget and capital efficiency, are gradually becoming more significant. Low server utilization is contrary to these concerns. In order to alleviate this utilization issue, data centers need to offer better storage resiliency, more flexible and scalable hardware architectures, and more sharing of devices between multiple servers to better utilize resources.

Solutions to these problems must be addressed both at the hardware level of bladed server systems, as well as in higher level software. In general, fixed, rigid architectures need to be replaced with dynamic architectures, offering flexibility in the allocation of resources amongst several systems in a bladed environment. One way to do this which has become popular in recent years is virtual machine (VM) technology which can efficiently virtualize computation and storage. Multiple virtual servers can be run on a single physical server, sharing CPU and memory resources. Storage virtualization solutions have also been developed allowing multiple VMs to share the same disk. I/O connectivity virtualization, however, has lagged behind these other two facets due to physical constraints in data center architectures. Applications are tied to a server and its designated I/O hardware, which results in an inflexible and inefficient infrastructure. Adapters, cables and network ports are physically assigned to specific servers. As a result, I/O performance acts as a bottleneck in advancing data center technology to the next level.

The future of I/O chip-to-chip interconnects lies with PCI Express (PCIe). PCIe has been around for several years, and has become the standard for local and chip-to-chip interconnects in bladed servers due to significant increases in bandwidth in recent years. It is also widely adopted for its efficiency, power, latency, simplicity, scalability from the legacy PCI standard, power and system cost advantages. However it does not yet support multi-root system architectures for managing system resources. With the next generation of systems relying on multiple processors for performance and increased operational efficiency we need improved support from PCIe hardware. This is essential in data centers which coordinate I/O resources amongst a large number of machines. In order to capitalize on the benefits of PCIe, the challenge lies in extending the standard to incorporate the requirements of a multi-root environment. The challenge is well-studied and several solutions have been proposed, including modifications to the non-transparent bridging (NTB) standard and Multi-Root I/O Virtualization (MR-IOV) extensions to the base PCIe specification. These proposals are convincing, but fail because of the large ecosystem of hardware support or levels of proprietary software development required (1).

A better approach lies in PCIe multi-root switches which offer a more straightforward alternative to the aforementioned solutions. The development of these switches this past year has opened up the door to a wide range of unexplored possibilities to improve the structure of bladed systems. Multi-root switches have a partitionable architecture with controls to configure multiple logical switches within a single device (1). And because of its recent emergence, it has yet to find its way through the development cycle to be integrated into platforms of today's blade and server systems.

This thesis investigates and experiments with an approach to integrate many features of a PCIe multi-root switch into next general multi-host platforms, allowing a single PCIe bus to be shared by several machines on the same bus. This in turn provides a means of dynamically allocating banks of I/O devices, such as Ethernet cards, Host Bus Adapters (HBAs) and graphics cards to different machines to provide them with additional bandwidth as needed. The hardware capabilities are now

readily available in various prototypes, but there is lagging support in software and operating system limitations that reduce its immediate practicality.

This thesis examines the feasibility and merit of using this new device in NetApp's mass data storage platforms, examining system requirements, performance costs, changes that need to be made to current systems, and a variety of other issues and concerns. We look to develop a methodology to add support for a switching solution during system operation. In particular, since NetApp's proprietary OS is built on top of FreeBSD, the thesis looks specifically at integrating multi-root switch features into this OS. The ultimate goal of this work is to derive a method to improve system flexibility, which have applications in improving resiliency, performance and efficiency of collections of servers in data clusters.

1.2. Objective and Scope

The objective of the thesis is to investigate the feasibility of integrating and reconfiguring a PCI Express multi-root switch in a multi-host environment on a FreeBSD platform in a live system. This thesis looks specifically at the mechanics of adapting a system and driver implementation needed to enable reconfiguration at the platform level. The project does not focus on more software-oriented issues such as dealing with packet loss which must be handled by the network protocols.

The investigation itself is broken down into several components. The first explores prerequisites to reconfiguration, studying the feasibility of implementation in the FreeBSD kernel and its PCI bus allocation algorithm. We conclude that some modifications are needed in the kernel (Chapter 5). Once these modifications are in place, we look at implementing the device driver module for the multi-root switch to handle reconfiguration (Chapter 5.2), and describe a series of tests of its functionality (Chapter 6). Finally, we verify that there is no significant performance degradation

with the use of the switch by comparing it to the expected hardware specifications of the switch (Chapter 7).

2. Background

In this chapter we present an overview of work that has been done with I/O connectivity in multi-host environments leading up to the development of multi-root switches.

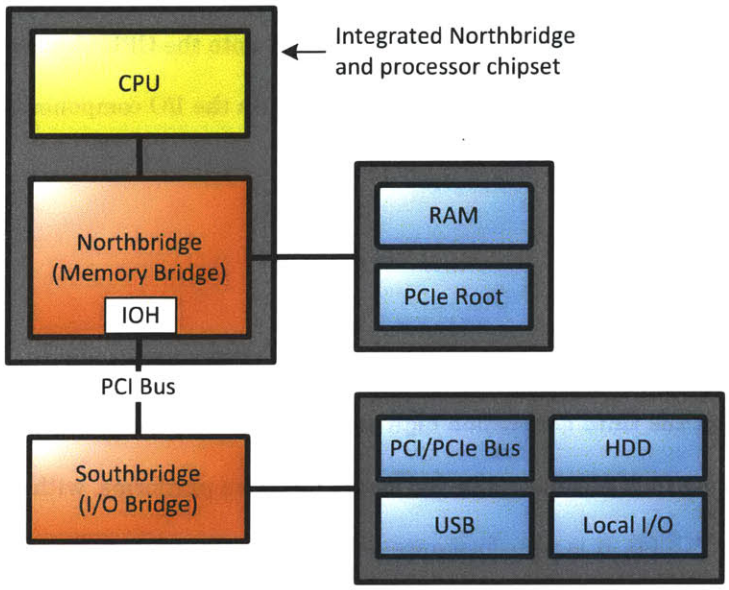


Figure 2-1: A typical Northbridge-Southbridge layout.

Figure 2-1 shows the general layout of the buses in a typical computer system. The isolation of the memory and processor from the I/O bus was designed to segregate rapidly changing components from the stable ones. Processor frequencies have increased continuously, and memory bandwidth has kept pace with the increase in processor speed. Historically I/O device speeds, however, have not increased at as significant a rate. This isolation allows the memory and processor to incorporate new design architectures while maintaining a common interface to the I/O bus. But as the demand for I/O bandwidth grows with the development of applications such as Gigabit Ethernet and InfiniBand, a newer standard that can offer scalable performance is essential. In these systems where high-performance I/O peripherals outstrip the bandwidth of modern Northbridge architectures, difficult tradeoffs need to be made (2). Over the last decade, PCI Express is the standard that has emerged as the front-runner to be adopted as the main I/O interconnect for all next generation systems. (3). But

in order to meet growing demands of the I/O bus, especially in an industry slowly gravitating towards large clusters, there are several challenges that need to be overcome.

Newer chipset architectures have gradually integrated the Northbridge onto the processor for improved performance to memory and the I/O bus. With Intel and AMD's most recently designed chips, all functions of the Northbridge are fully integrated onto the CPU chip as illustrated in *Figure 2-1*. The I/O Hub (IOH) acts as the connection point between the I/O components and the processor(s) through the high performance PCI Express interface.

2.1. PCI Express

PCI Express (PCIe) is the third generation high performance I/O bus used to interconnect peripheral devices in applications such as computing and communication platforms, with a foundation rooted in its predecessor bus, PCI (4). It is a performance-oriented interface protocol that improves bus performance, while reducing overall system cost, taking advantage of predecessor buses and also new architectural developments. Its predecessors are parallel interconnects with multiple devices sharing the same bus. PCIe on the other hand is a serial, point-to-point interconnect that allows for communication between two devices via a switch. This provides scalability as a system and can consist of a large number of devices connected via a hierarchy of switches.

Using a point-to-point interconnect allows transmission and reception frequencies to scale to higher levels than previously achieved. PCIe Generation 1 transmissions have a data rate of 2.5 GB/s, Gen 2 PCIe can reach a rate of 5GB/s, and Gen 3 up to 8GB/s. PCIe also encodes transactions using a packet based protocol, where packets are transmitted and received serially along each PCIe lane, lending itself to a scalable increase in bandwidth as the number of lanes in a link increases.

PCI Express's foundation and compatibility with older standards, coupled with its respectable performance boost, flexible architecture and scalable bandwidth makes it a standard that will be adopted in next generation systems. Unfortunately, the standard PCIe specification offers limited support for multi-processor configurations. As such configurations become more and more prevalent, this deficiency needs to be remedied. Several solutions have been implemented to offer support for multi-root systems, but these solutions each have their respective constraints.

2.2. Multi-Root System Solutions

There are two common solutions to offering multi-root functionality with PCI Express. The first involves making modifications to the non-transparent bridging (NTB) standard and the second is a Multi-Root I/O Virtualization (MR-IOV) extension to the basic PCIe specification. Both of these have merit, but present a number of challenges associated with integrating PCI express into today's bladed systems. In particular, the complex requirements for software and hardware support and proprietary implementations have restricted their widespread use.

2.2.1. Non-Transparent Bridging (NTB)

PCI systems have long been implemented with multi-root processing, even though it was originally designed as an interconnect for personal computers. Because of the architecture at the time, protocol architects did not anticipate the need for multiprocessors. Architects designed the system with the assumption that the single host processor would enumerate the entire memory space. With multi-core systems, this enumeration would fail as multiple processors would attempt to service the same system requests (5).

Non-transparent bridges are a means for PCI systems to handle multiple processors. The bridge acts as the gateway between the local subsystem and another subsystem, isolating subsystems from one

another by terminating discovery operations. These subsystems may include multiple processors or machines. The bridge can segregate two endpoints, preventing discovery of one endpoint from the other, while creating apertures into the address space of the endpoint on the other side of the bridge through address translation. Alternatively, bridges can hide an endpoint from a processor, and a processor from an endpoint, thus allowing multiple processors to share the enumeration process. Packets intended to reach one of the endpoint through a bridge requires the bridge to translate the address into a different address domain in order to forward the packet onwards. In the opposing direction, translation is also needed between the local address space and the global address space (5). NTB can be integrated in the processor or on a switch. This implementation was ported to PCI Express, allowing for NTBs between PCI, PCI-X and PCI Express connections.

For dual-host applications, NTB offers all the necessary functionality in multi-root systems. But unfortunately in these multi-root bladed servers, the non-transparent barrier is a nuisance that must be handled delicately in software. PCIe needs to directly configure I/O endpoint devices, and the increased complexity of address translations is a serious restriction and has a substantial overhead. Additionally, it results in a rigid, non-scalable platform (6). These concerns make it difficult to effectively use NTB in a data center environment.

2.2.2. I/O Virtualization

I/O Virtualization (IOV) is a set of general PCI Express based hardware standards developed by the PCI-SIG committee. There are two variants of I/O Virtualization: Single Root I/O Virtualization (SR-IOV) and Multi-Root I/O Virtualization (MR-IOV). The specification for MR-IOV was finalized in 2008 (7).

SR-IOV supports existing PCI Express topologies that have a single root complex. The standard specifies how multiple guest operating systems (OS's) running independently on a single physical

server with a single PCIe root can share the same physical I/O devices. Resources connected to a PCIe endpoint can be shared among several System Images (SI's) through the Virtualization Intermediary (VI). This standard only supports a single CPU entity. The MR-IOV standard is extended from the SR-IOV concept. It can support multiple CPU entities allowing a hierarchy of PCIe devices to be shared among different physical servers. Independent servers with separate PCIe roots are connected to a pool of shared I/O devices through a MR-IOV switch. The servers and I/O devices are entirely decoupled and can scale independently. MR-IOV has the potential to leverage the performance of PCIe, where a high I/O throughput can be achieved independently per blade. It also reduces power, improves device efficiency, and increases flexible configuration of I/O devices. All of these benefits make it a very recognized standard in the industry.

Since both SR-IOV and MR-IOV are incorporated in hardware, they require explicit support from I/O device. Specifically in the case of MR-IOV, it has yet to gain popularity and support. Unfortunately, this means that it will be some number of years before MR-IOV will begin to play a more significant role in I/O virtualization. The high cost of developing devices that have the capability to take advantage of MR-IOV discourages research in the area. And conversely, with little research in this area, there is no immediate driving force for companies to develop these expensive devices. MR-IOV is ideal for bladed server systems. But, the lack of demand and availability of MR-IOV capable devices and the time needed to establish the standard in the industry does little to alleviate the needs for a multi-host solution in today's bladed server systems (8).

2.2.3. PCI Express Multi-Root Switch

The PCI Express multi-root switch is a third alternative to solving the multi-root architecture problem. Though, like NTB and I/O virtualization, it does have its shortcomings.

2.3. PCI Express Switch

2.3.1. Standard PCI Express Switch

The PCI Express switch provides a PCI connection between multiple I/O devices, allowing fan-out to multiple peripherals and aggregation of endpoint traffic. Within each switch one port is designated the upstream port and the others downstream. This device improves provisioning of Northbridge resources such as memory, bandwidth and connectivity. Limited Northbridge resources can be distributed to multiple I/O endpoints, to make better use of bandwidth. It provides a means for load balancing, allows I/O devices with smaller bandwidth requirements to share a north bridge link, and alternatively, if a port is oversubscribed, downstream ports can efficiently allocate the device the full available bandwidth. Non-sustained traffic patterns can be controlled and managed through the switches (2).

The drawbacks of inserting a multi-level hierarchy of PCIe switches into the interconnect architecture are increased latency, system cost and required board space. Effective switch architecture optimized for a specific I/O peripheral would help to mitigate some of these concerns, but this fails to provide a unified solution to the problem. The more elegant solution lies with multi-root switches.

2.3.2. PCI Express Multi-Root Switch

An alternate approach to offering multi-host functionality is through a PCI Express multi-root switch. Like the standard PCI Express switch it acts as the intermediary between the Northbridge host and the endpoints. This switch can be partitioned into two or more isolated logical domains each accessible by a single host or root complex. It functions exactly like a hierarchy of multiple standard PCIe switches, but it allows multiple hosts to be connected to their I/O devices over a common

backplane, as illustrated in *Figure 2-2*. With this shared interface, the architecture helps to reduce cost, power and board area, and offers a platform with fan-out capability as well as high performance I/O (3). Improved flexibility and system resiliency is also available if partitions are configurable. Switch partitioning offers many of the advantages of MR-IOV, but ultimately, its main drawback is that it does not allow for I/O device sharing. NTB is still required for communication between hosts (5).

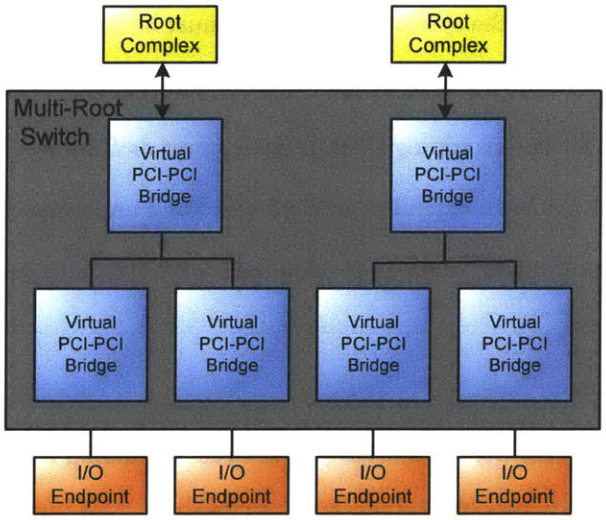


Figure 2-2: Multi-root switch partitioning.

Since this partitionable switch is a hardware extension to the standard PCI, it retains support for older, lower bandwidth PCI devices as well as the faster PCIe devices. It does not require a full hardware migration, as is required of MR-IOV, to profit from these benefits. Instead, it breaks the PCIe switching protocol down to the basic components in order to leverage the simplest of transactions across the switch since each virtual partition is logically discrete (9). In each hierarchy, its configuration, switching operation, and reset logic are isolated from the others. Transactions are passed across virtual PCI-to-PCI bridges associated with each logical partition.

Replication of the control and management structures allows any virtual PCI-to-PCI bridge to be associated with an established virtual bus, thus increasing architectural flexibility. This association can take place either statically at the time of a fundamental reset to the switch, or dynamically while

the switch is operating. However, in order to enable dynamic reconfiguration of the PCIe hierarchy, it needs to draw on similarities from the well-established PCI hot-plug standard.

2.4. PCI Hot-plug

Hot-plug is a system for managing devices that can be dynamically attached to and removed from a system without powering down the system. The system must be designed such that it can detect and respond to changes in the hardware accordingly, without interrupting regular operations. Currently, hot-plug is the standard for hot-swapping USB devices, input devices, and IEEE 1394 Firewire devices. In some cases, it has been extended for high-end SCSI and some common network interfaces, however in many cases, vendors develop their own proprietary methods, ignoring the standard.

In a server environment where high-availability is desired, extending Hot-plug for PCI and PCI Express devices is ideal. As originally designed, the PCI bus was not intended to support installation or removal of PCI devices while power is applied to the machine. Doing so may result in damage to components and a confused OS (10). But as the PCI specification evolved, it became obvious that this was an inconvenience that needed to be resolved. In June 2001, PCI-SIG released Version 1.1 of the PCI Hot-plug Specifications whose primary objective "is to enable higher availability of file and application servers by standardizing key aspects of removing and installing PCI add-in cards while the system is running" (11). Currently, the Hot-plug standard, although it does exist in experimental implementation, is not officially supported on any platform.

PCI and PCCard Hot-plug were quickly adopted as a standard feature by GNU/Linux 2.4 kernel series, and became more prevalent in the 2.6 series, available to most busses and driver classes (12). Unfortunately, FreeBSD does not. While it offers hot-plug for some busses, such as USB and Firewire, FreeBSD currently lacks full support for hot-plug PCI devices (13). There is limited

support in the underlying infrastructure, and also a lack of necessary device drivers. In order for multi-root PCI Express switches to dynamically accommodate changes in the virtual partitions in a FreeBSD environment, an alternative approach is needed.

3. PCI Express Operation and Specifications

This chapter describes additional details about the operation of PCIe. We will provide a high level overview of the PCI bus system topology, the Configuration Space of PCI devices according to the PCIe specifications, transaction routing protocols within the bus and the PCI bus enumeration process on system startup.

3.1. System Configuration - PCI Bus

The two main divisions of a chipset include the Northbridge and the Southbridge. The Northbridge connects the host processor bus to the root PCI bus, and the Southbridge connects the root of the PCI bus to I/O devices, including PCI devices, the USB controller and DMA controller.

A host domain can support up to 256 buses, numbered from 0 to 255. Each bus can support up to 32 devices, numbered from 0 to 31. And each device can support up to 8 functions, where each function is a logical device, numbered from 0 to 7. Each of the functions within a device provides a stand-alone functionality, for example, one function could be a graphics controller, while another might be a network interface.

There are three major components to a PCIe system: the root complex, switch and endpoint, shown in *Figure 3-1.*

1. Root complex: The root complex is the device that connects the CPU and the memory subsystem to the PCIe fabric. It acts as the barrier for all requester or completer transactions passed between the processor and the PCI hierarchy, generating memory and I/O requests on behalf of the CPU, and also passing transactions from the PCI fabric out to the processor. It implements all central resources, including the hot plug controller, power management controller, interrupt

controller, error detection and reporting logic. Bus 0 of the hierarchy is the internal virtual bus within the root complex, and it is initialized with a device number of 0 and a function number of 0. A root complex can have multiple ports with which it can transmit packets in and out of.

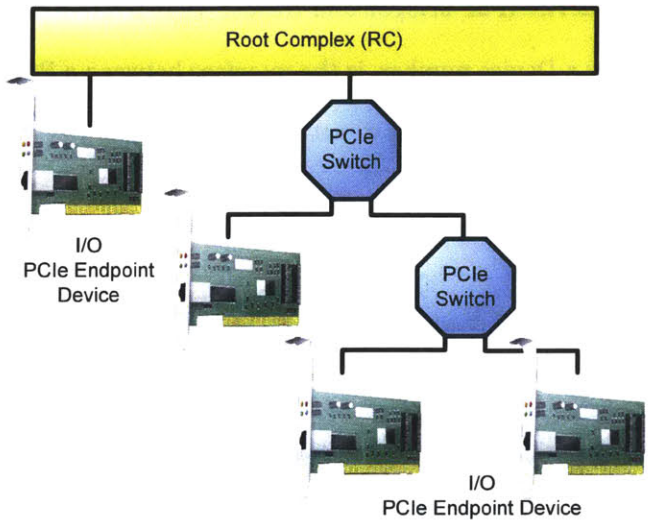


Figure 3-1: PCI bus topology

2. Endpoint: Endpoints are I/O peripherals, such as Ethernet, USB or graphic devices. Like the root complex, endpoints are requesters or completers of PCIe transactions. They are identified by a bus, device and function number, and implement Type 0 PCI configuration headers.

3. Switch: Switches consist of two or more logical PCI-to-PCI bridges connecting PCI components together, where each bridge is associated with a switch port. One port points towards the root, and the other ports point away. A switch implements Type 1 PCI configuration headers. The configuration information includes registers used for packet routing and forwarding which enables transactions to find their way to their destinations.

Other Terminology:

Lane: A serial point-to-point connection between two PCI Express devices.

Link: A link consists of a number of lanes. A connection can have link widths of x1, x2, x4, x8, x12 x16 or x32.

Header Type: There are three types of PCI devices functions, each implementing a different configuration space. Header Type 0 devices are endpoint PCI devices, Header Type 1 refers a PCI-to-PCI Bridge device (P2P bridge), and Header Type 2 refers to a PCI-to-Cardbus bridge.

Port: A port, identified by a Device number, is the interface between a PCIe component and the Link. An upstream port points in the direction of the root complex. And a downstream port points away.

3.2. Configuration Space Registers (CSR)

The configuration space for each PCI endpoint is mapped to the system address map. During boot-up the BIOS initializes the PCIe configuration space, allocating it a 256MB window, in other words a 4KB configuration space for each endpoint (8 functions x 32 devices x 256 buses x 4KB of space = 256M). This address is then passed from the BIOS to the OS.

Every PCI function must implement PCI configuration space within its PCI configuration registers. Each PCI function possesses a block of 64 configuration dwords reserved for the configuration space registers (CSR's). The first 16 dwords are detailed in the PCI specifications, and referred to as the Configuration Header Region or Header Space. The remaining 48 dwords are device specific. See *Figure 3-2* for a Header type 0 and 1 CSR.

3.3. Transaction Routing

There are four transaction types that can be requested by the Transaction Layer, each of which accesses a different address space. Memory Reads/Writes are used to transfer data from/to a location in the system memory map. I/O Reads/Writes transfer data from/to the system I/O map.

Configuration Reads/Writes transfer data from/to the configuration space of a PCI device. And Message transactions provide vendor specific general messaging and event reporting options.

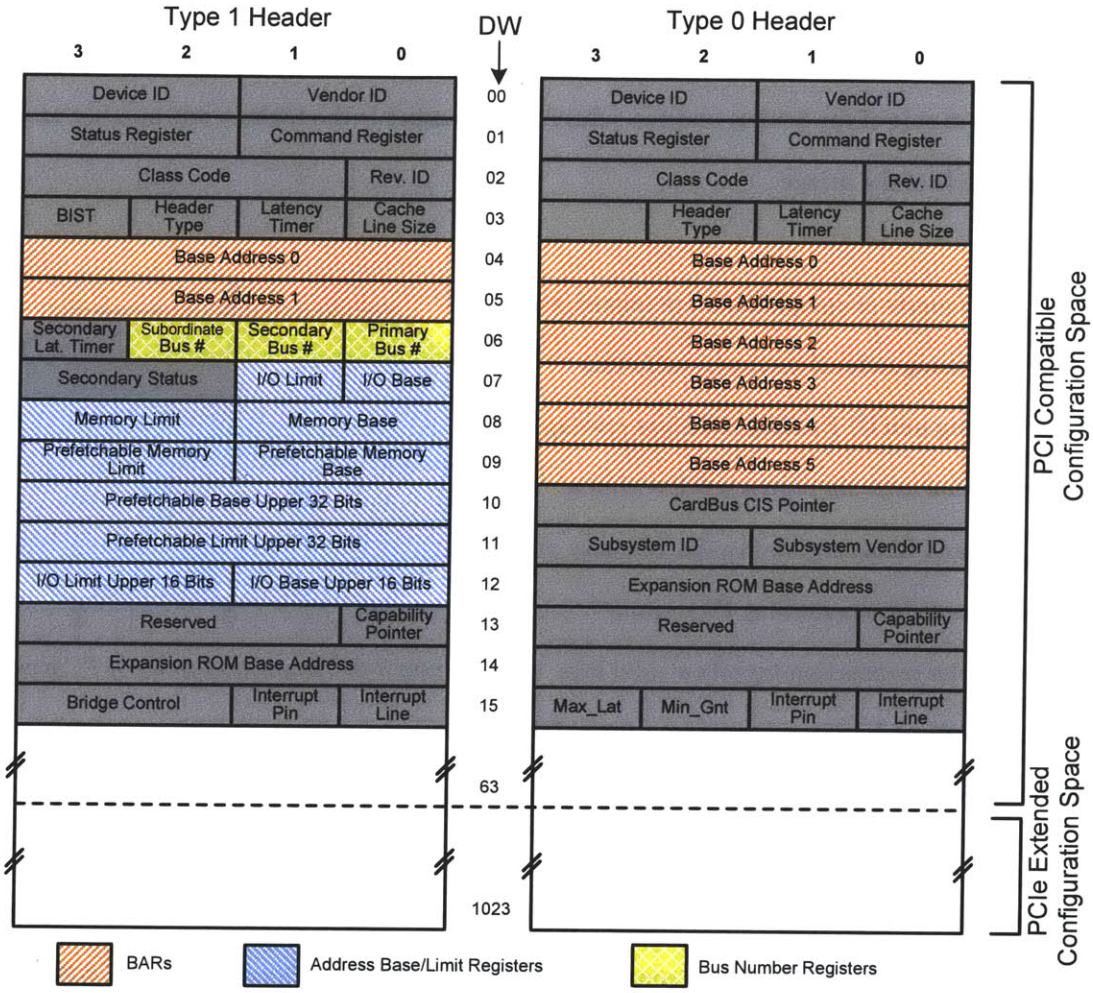


Figure 3-2: PCI Configuration Space for Type 1 and 0 Headers.

Several sets of registers are used for transaction routing as shown in *Figure 3-2*: BARs, Base/Limit address pairs and bus number registers. Firstly, the Base Address Registers (BARs) are implemented by devices that require system memory or system IO. They are programmed with the start addresses for a range of address that the device may respond to. Secondly, are the Base/Limit register pairs supported in Type 1 headers. The IO base and IO Limit pair defines the start and end address of the range addressable by transactions to be forwarded to the secondary bus. If these registers are not implemented, the primary bus ignores the I/O transaction. Likewise, the same

holds for the Memory Base (membase) and Memory Limit (memlimit) registers for memory transactions. And lastly, are the three bus number registers in Type 1 Headers. The primary, secondary and subordinate numbers determine the bus range of a PCI branch. The primary bus is the bus number of the upstream side of the bridge, the secondary bus is the bus number connected to the downstream side of the bridge, and the subordinate number is the highest bus number on the downstream side of the bridge.

Transaction requests or completions are tagged, and can be routed using several methods, including Address Routing and ID Routing. Address routing requires that downstream requests fall within a bridge's Base and Limit register range. Memory and I/O requests use address routing. ID routing requires that the transaction targets a valid bus number in the secondary to subordinate range of the bridge. Completions and configuration requests use ID routing.

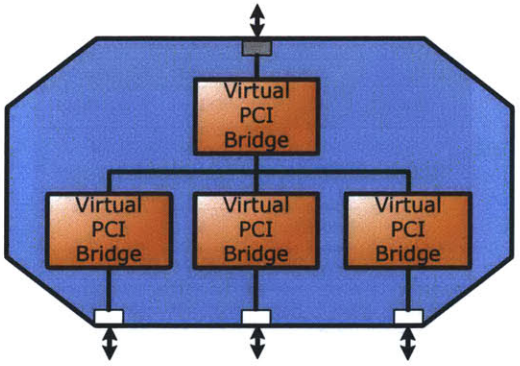


Figure 3-3: Virtual hierarchy of PCIe switch.

The MR switch is a hierarchy of PCI-to-PCI bridges with one upstream PCI-to-PCI bridge and one or more downstream PCI-to-PCI bridges connected by an internal virtual bus (See *Figure 3-3*). The upstream bridge is assessed by Type 0 configuration requests. The downstream devices are accessed in two ways. If the request targets the upstream bridge's secondary bus number (the virtual bus), the Type 1 request is converted to a Type 0 request. If the target request is within the secondary to subordinate bus number range, the request is forwarded out of the switch unmodified. If not within this valid range, the request is terminated.

3.4. PCI Bus Enumeration

When a system is first booted up, the configuration software has not yet scanned the PCI Express fabric to discover the machine topology and how the fabric is populated. The configuration software is only aware of the existence of the Host/PCI bridge within the root complex and that bus number 0 is directly connected to the downstream side of the bridge. It has not yet scanned bus 0 and therefore does not know how many PCIe ports are implemented on the root complex. The process of scanning the PCIe fabric to discover its topology is referred to as the enumeration process.

The PC's BIOS, and not the OS, performs the initial PCIe bus enumeration, performing configuration reads and writes to set the BARs, resource base and limit ranges and bus number registers. The OS follows suit, re-scanning the PCI bus to read the registers set by the BIOS to build the OS specific PCI data structures and to reserve the assigned resources.

The actual enumeration process performs a depth-first search of the PCI hierarchy. The enumerating software probes each of the 256 buses, and for each of the buses probes all 32 possible devices. When a device is discovered, the software performs a depth-first search to scan for additional functions/devices on the child bus.

4. Prototype Hardware Configuration

In this chapter, we describe all the hardware used in the thesis. The setup involved two host machines, two PCIe multi-root (MR) switches, several endpoint devices for various test scenarios, and several helper devices used for verification and analysis.

4.1. Host Systems

The system was setup with two host machines as the two root complexes connected to the MR switch. Both host machines use an Intel prototyping platform with a dual processor to single I/O hub topology. The I/O hub sits in the Northbridge integrated onto the processor chipset and connects to the Southbridge via a PCIe bus. The systems use the Greencity motherboard with dual Xeon 5500 Series processors and a 5520 Chipset I/O Hub (14). The development platform in use is 64-bit FreeBSD 8.1.

4.2. MR Switch Setup

Prototype MR switches from two different vendors: PLX Technology and IDT were tested. The next sections look at the specifications and setup for each of the MR switches.

4.2.1. PLX Setup

PLX Technology's ExpressLane PEX 8664 is a fully non-blocking, low-latency, low-cost and low-power 64 Lane, 16 Port PCI Express Gen 2 Multi-Root Switch, full compliant with the PCI Express Base R2.0 specifications. The PEX 8664 allows users to configure the switch in two modes: conventional base mode and Virtual Switch mode. In base mode, the PEX 8664 acts as a standard

PCIe switch, supporting a single Host hierarchy. In Virtual Switch (VS) mode it supports up to five Hosts, creating five virtual switches within the 8664, each with its own virtual hierarchy (15). For the purposes of the project, the PEX 8664 was used solely in VS mode. There are many possible port configurations of the 16 ports where each port can be freely configured as an upstream or downstream port. The bandwidth of each port is also variable, and can be set to link widths between x1 and x16. *Figure 4-1* shows the physical setup of the PLX MR switch.

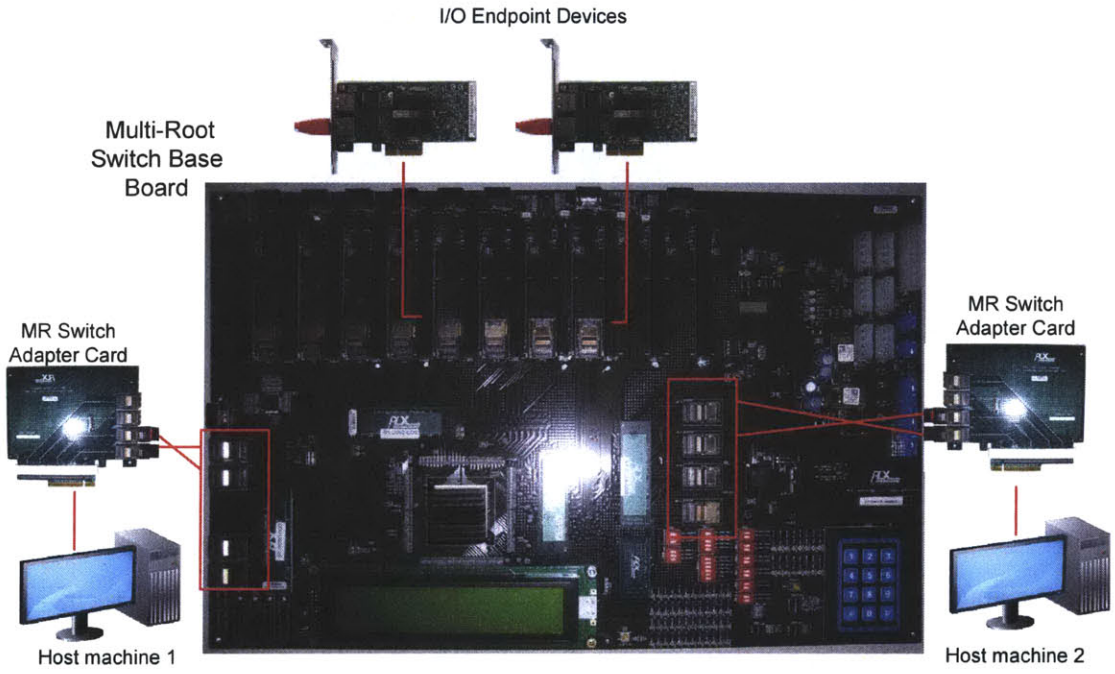


Figure 4-1: PLX MR Switch setup with two hosts.

4.2.2. IDT Setup

The 89HPES48H12 is a member of the IDT PRECISE family of PCI Express switching solutions. It is a 48-lane, 12-port system PCI Express interconnect switch offering high throughput, low latency, and a simple board layout with a minimum number of board layers. It provides 192 Gbps of aggregated full-duplex switching capacity through 48 integrated serial lanes. Each lane provides 2.5 Gbps of bandwidth in both directions and is fully compliant with PCI Express Base Specification Revision 1.1 (16). *Figure 4-2* shows the physical setup of the IDT MR Switch.

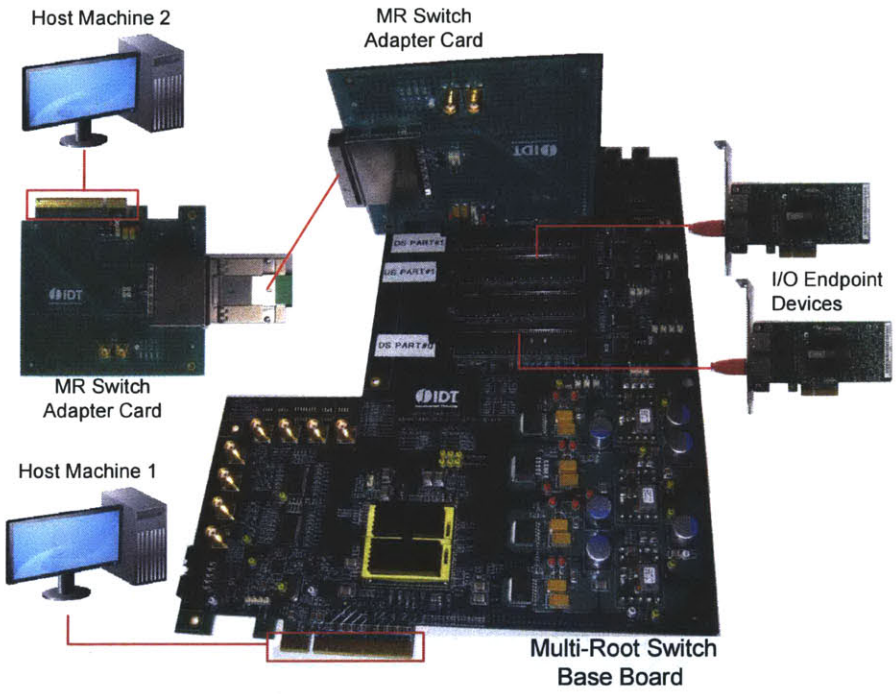


Figure 4-2: IDT MR Switch setup with two hosts.

4.2.3. Multi-Root Switch Selection

Prototype multi-root switches from PLX Technology and IDT were setup according to their specifications. Even though they both provide the same fan-out and peer-to-peer communication functionality, their implementation and requirements differ in some regards. These differences influence the adoptability and usability of the device in the specified test environment, restricted by both system hardware and software. Using the resources available at NetApp at the time, the PLX PEX 8664 was the more acceptable choice for additional testing. The Rapid Development Kit (RDK) was more compatible with the provided hardware, and the virtual switch and partition configurations more easily assessed and configurable in FreeBSD. As a result, for the remainder of the thesis, all code and tests were run using the PEX 8664 prototype board.

4.3. Endpoint devices

Two PCIe endpoint devices were used for testing: generic Intel Ethernet cards, and NetApp's NVRAM8 card.

4.3.1. Intel Ethernet Card

The Intel 82571EB Gigabit Ethernet Controller uses the PCI Express architecture (Rev 1.0a), and incorporates two fully integrated Gigabit Ethernet Media Access Control (MAC) and physical layer (PHY) ports via two IEEE 802.3 Ethernet interfaces (17). This endpoint requires a relatively small amount of memory and I/O space to be allocated to it by the host system during initialization. The small memory and I/O requirements make it an ideal candidate for testing. It provides a simple case for moving an endpoint from one virtual partition to another by reconfiguring the MR switch (See Chapter 6.2 for details about the test).

4.3.2. NetApp NVRAM8 Card

NetApp's proprietary NVRAM8 card contains a DMA engine that can generate packet traffic through the PCIe fabric. This device provides a means of sending controlled traffic through the switch and upstream to the host root complex which can be monitored using NetApp-developed performance measurement tools and PCI bus analyzers. *Figure 4-3* shows a picture of the NVRAM8 card.

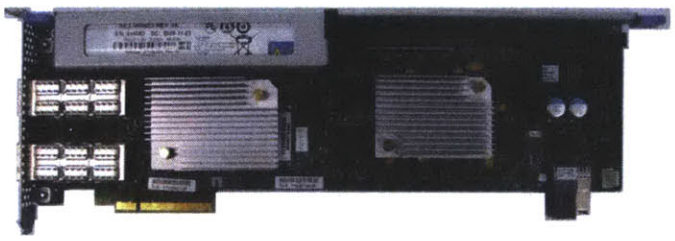


Figure 4-3: NetApp's NVRAM8 card.

4.4. Miscellaneous devices

4.4.1. Aardvark I2C connection – EEPROM access

Both PLX and IDT switches provide a means of switch configuration through an EEPROM and I2C interface. Access to the registers allow for manual configuration of the Virtual Switch (VS) partitioning, port assignment to partitions, monitoring of VS links and global PEX 8664 registers. After reprogramming the EEPROM system restart is required for the changes to take effect. In order to program the EEPROM an Aardvark 12C/SPI Host Adapter was used. The Host Adapter is a fast I2C/SPI bus to USB adapter that transfers serial messages using the I2C and SPI protocol. The adapter is connected to an external computer running an SDK enabling the reading and writing of EEPROM registers. *Figure 4-4* shows the adapter setup.

Figure 4-4: Aardvark I2C/SPI Adapter setup for MR Switch EEPROM access.

4.4.2. VMETRO Network Bus Analyzer

VMetro's Vanguard Express Analyzer with Host exerciser is a free RDK for developers of PCI Express, PCI-X and PCI systems (18). The Host Exerciser is a powerful utility that allows users to generate PCIe, PCI-X or PCI traffic, from the Host's root complex or Host Bridge to the endpoint VMETRO Analyzer. The Exerciser is controlled by VMETRO's BusView application software either running on the same machine or a remote machine via an Ethernet connection. BusView offers a GUI on Window systems. It allows a user to capture trace files from the VMETRO analyzer, generate

Host Exerciser scripts which can detail a specific series of transactions, scan the configuration space, perform memory testing and many other tools. This package provides a controlled means of testing the performance of the MR switch. An image of the analyzer is shown in *Figure 4-5*.

Figure 4-5: VMetro's PCI Bus Analyzer

5. Software

In this chapter, we look at the software component of the thesis. There are two areas of consideration. The first is to make modifications to the bus enumeration and resource allocation process to provide a platform for reconfiguration. The second component is a device driver for the multi-root (MR) switch to implement various reconfiguration operations.

5.1. PCI Bus Space Allocation and Enumeration

When a multi-root switch is reconfigured, the PCIe bus and device resources, including memory and I/O space, must in turn be reconfigured. As described in Chapter 3.4, the enumeration of the bus and resource allocation is assigned by the BIOS and OS software at boot-up. But if the MR switch is re-configured after boot-up, the PCI bus must be able to accommodate for the addition or removal of I/O endpoints. This requires that the bus enumeration process and resource allocation method be modified. Several approaches are possible, making modifications either at 1) the firmware level, 2) the OS level or 3) the MR Switch device driver level.

5.1.1. Method 1: Firmware Configurations

Since PCIe bus enumeration and device resource allocation is initially performed by the BIOS, it would be ideal if that process could be modified here. Unfortunately, the standard BIOS was not written in a way that makes it possible to add configurability. It assumes that the hardware configuration will remain static after the system is booted. A bus number is assigned to each root port or switch virtual bus at boot-time. If a switch or endpoint is added to the hierarchy after the system boots, there will be insufficient bus numbers to initialize the new components since the range of bus numbers assigned to a branch must be contiguous between the secondary and subordinate bus numbers. Similarly, with device resources, memory and I/O space are only allocated for populated slots, and in order for resources to be allocated for a new device, its memory and I/O space must also

be assigned contiguously with the rest of the branch. Hence complications arise when a new device is inserted, or if an old device is replaced with one that requires more resources.

Two modifications need to be made to the BIOS. First, it needs to introduce gaps in the PCI bus enumeration. Inserting gaps in bus numbers as a pre-emptive measure would allow for new devices to be inserted adjacent to devices existing in the hierarchy at boot-up. Secondly, forming partitions in memory and I/O resources would leave resources available for allocation to new devices. Assigned ranges of memory and I/O could be pre-allocated for the MR switch. On proprietary platforms, such as NetApp's filers, where full control over the firmware is readily available, this is a feasible approach. But for a more generic, more widely adoptable environment, such as the standard PC setup used in this thesis, this is not possible, and alternate methods are required.

5.1.2. Method 2: FreeBSD Kernel Configurations

After the boot-up process is completed by the platform's BIOS, the OS software takes over. At this point, there are two possible approaches. The first is to proceed with the standard enumeration process. When reconfiguration is required the OS must stop all devices sharing the same root port as the new device. Once all the devices are in a quiescent state, the branch of the hierarchy extending from this particular root port, can be re-enumerated and re-allocated new memory and/or I/O resources. Once the reassignment is complete, the OS can restart all the device drivers along this branch. This is a good solution if multiple devices are connected or disconnected at the same time. In scenarios where an individual device is attached and detached, or if a device is frequently added/removed, then this approach is highly inefficient. A further complication arises if a device along the branch cannot be put into a quiescent state, preventing reconfiguration.

The second approach is for the OS to re-enumerate the PCIe bus, simply overriding the assignments derived by the BIOS. The OS would pre-allocate PCI bus space and resources to root ports,

anticipating the addition of new devices. Since the exact configuration is unknown, the OS must account for the most extreme case, reserving resources to support the maximum system configuration on all root ports. Naturally, the advantage of this approach is that it does not require an entire branch of devices to be stopped. It is also better suited for applications when the maximum configurations are known in advance. The drawback of this approach is that resource space is wasted unless the maximum configurations are achieved. Introducing a fixed gap at each root port reduces the bus and resource space by a constant factor. But in systems where resource usage is sparse, such as NetApp's filers which have large pools of memory and I/O space, this approach is a practical one.

Figure 5-1: Pseudocode of modifications made to pci_pci.c.

```
    Global parameters
        - busnum = first unallocated bus number
        - bus_gap_size = size of the gap
    Local parameters
        - dev = the probed device (a bridge) to be attached
        - parent_secbus/subbus = secondary/subordinate bus of dev's parent bridge
                                 = parent(parent(dev))
        - init_pribus/secbus/subbus = initial primary/secondary/subordinate of dev

1   void pcib_attach_common(device_t dev)
2       ...
3       Call pcib_update_busnum(dev) to insert gaps
4       ...
```

```
1   static int pcib_update_busnum (device_t dev)
2       struct pcib_softc sc ← device_get_softc(dev);
3       if (busnum + bus_gap_size + 1 >= 0xff)
4           return error, exceeded max number of buses
5       if dev connected to root bus, sc->pribus == 0
6           Set sc->secbus to sc->subbus range to the bus_gap_size
7           Increment busnum by bus_gap size to set it to next available bus number
8       else
9           device_t parent ← retrive handle on parent bridge of dev = parent(parent(dev))
10          Update sc->secbus ← parent_secbus + (init_secbus - init_pribus)
11          Update sc->subbus ← parent_secbus + (init_secbus - init_pribus)
12                                            + (init_subbus - init_secbus)
13          if (sc->secbus <= parent_secbus or sc->subbus > parent_subbus)
14              return error, subbus exceed legal bus range
```

To test the practicality of the second approach, it was implemented with modifications made to the FreeBSD kernel in the PCI bridge file `pci_pci.c` as shown in *Figure 5-1*. The entry point to the gap insertion function, `pcib_update_busnum`, is in the FreeBSD PCI bridge function, `pcib_attach_common`. When the attach function is called, `pcib_update_busnum` is first called to

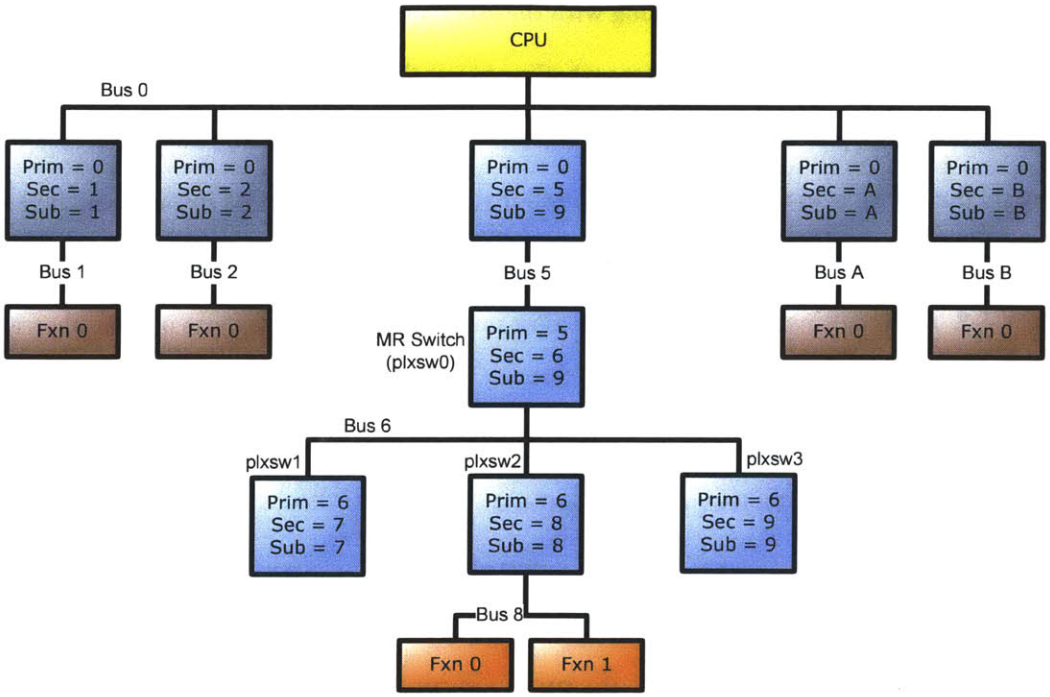


Figure 5-2: Original PCI bus hierarchy.

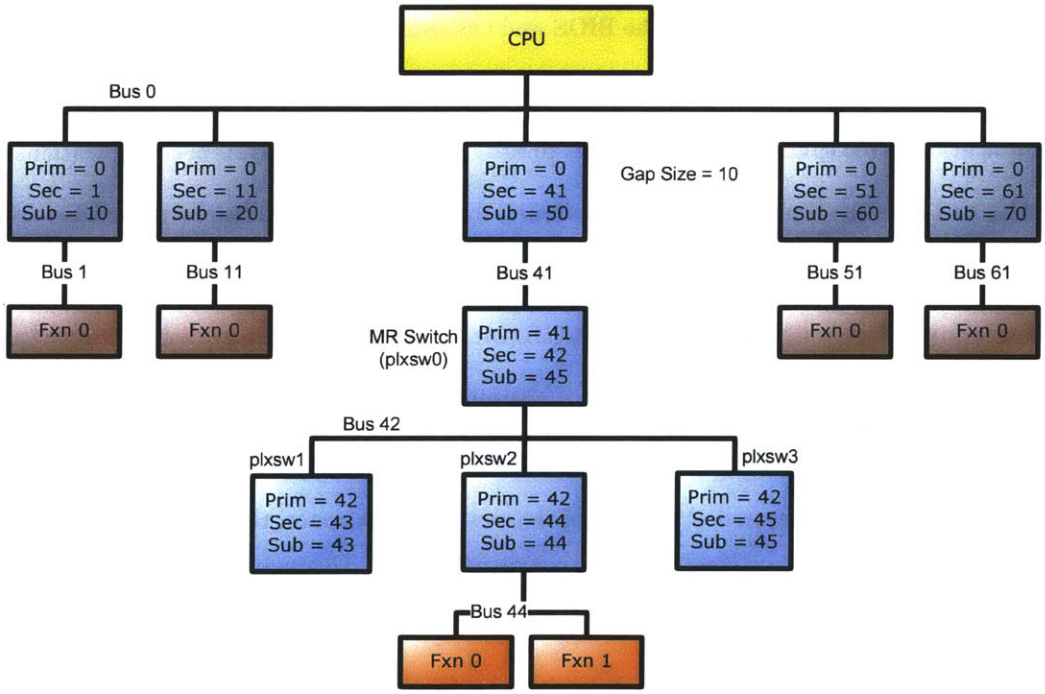


Figure 5-3: PCI bus hierarchy with modified enumeration algorithm.

update the bus numbers before proceeding with the standard attach implementation. In `pcib_update_busnum`, Line 5 determines whether or not the probed device is connected directly to a root port. If it is, inserts a gap between the secondary and subordinate bus number (Lines 6-7). Otherwise, the device is in the downstream hierarchy, and should retain its initial secondary and subordinate bus offsets (Lines 9-12). The original PCI enumeration of the system hierarchy is shown in *Figure 5-2*. The resulting PCI enumeration after introducing a fixed gap size of 10 is shown in *Figure 5-3*.

5.1.3. Method 3: Device Driver Configurations

Reconfiguration can also be handled by the MR switch device driver itself but this approach has many restrictions since it is applied after the BIOS and OS initializations and was not implemented in the thesis. To ensure that there is enough bus space, the unused bus numbers at the ends of the resource space can be exploited. Since the BIOS and OS enumerates devices sequentially, using the next available bus number, the bus numbers after the last enumerated device may be unassigned. In some applications, this space may be sufficient to satisfy the maximum system configurations of the MR switch and thus the branch of devices stemming from the MR switch can be moved here. Once re-allocated to this unassigned space, endpoints can be added and modified freely. But this implementation is highly dependent on the specific hierarchy of the entire system. For example, some BIOS implementations prevent the use of bus numbers beyond the maximum number discovered during initial enumeration. In another case, some devices, integrated into the system and OS, are intentionally enumerated at the end of the available bus space, thus rendering this approach ineffective. In general, many system-specific factors hinder the practicality of this approach.

In a relatively static system, one possible workaround is to designate a constant number as the first enumerated bus of the MR switch, but this limits flexibility. Another workaround is to provide a patch to the kernel such that when a particular MR switch device is probed during OS enumeration,

fixed resource gaps are allocated for that particular root port. This solution allows for better use of the bus and resource space. But these approaches all require a platform and device-specific implementation, which does not help to make the use of MR switches more easily adoptable and flexible.

5.2. MR Switch Device Driver Development

Assuming the PCI bus is capable of accommodating reconfiguration using one of the methods described in the previous section, the actual reconfiguration process was implemented. To integrate MR Switch dynamic reconfiguration into the system, a device driver for the PEX 8664 was written and loaded into the FreeBSD kernel. The driver handles the process of reconfiguring the switch, managing bus and memory allocation to support the addition, removal, suspension and resumption of devices in the live system.

5.2.1. Device configurations

The process of dynamic reconfiguration begins with the hardware. The registers stored on the MR switch are used to alter the state of the device. A subset of these registers control the partitioning of the virtual switches. These must first be set to remove hardware flags, allowing the host systems to detect a physical connection from root complex to endpoint and to then be able to probe and attach the enabled ports. Depending on the nature of the reconfiguration, the appropriate registers are set. The set of registers is shown in Table 1 (15).

Table 1: PLX PEX 8664 configuration registers (15).

Register	Offset	Description
Virtual Switch Enable	358h[4:0]	The register's VSx Enable bits are used to enable or disable virtual switches within the system. There is one bit per virtual switch (VS0 through VS4). Setting/Clearing a bit enables/disables the corresponding virtual switch. Bit [0] = Enable VS0 ... Bit [4] = Enable VS4
VSx Upstream	360h – 370h	These registers define the upstream Port of each virtual switch. There is one register per virtual switch (VS0 through VS4). Bits [4, 2:0] define which Ports are the singular upstream Ports, within the corresponding virtual switch.

VSx Port Vector	380h – 390h	These registers define the upstream and downstream Ports within each virtual switch. There is one register per virtual switch (VS0 through VS4). Each register has one bit per PEX 8664 Port. Bits [23:16, 7:0] correspond to Ports 23 through 16 and 7 through 0, respectively.
Port Configuration	300h	Configures the link widths of the upstream ports. Bits [1:0] Port configuration for Station 0 Bits [3:2] Port configuration for Station 1 Bits [9:8] Port configuration for Station 4 Bits [11:10] Port configuration for Station 5 00b = x4, x4, x4, x4 01b = x16 10b = x8, x8 11b = x8, x4, x4

5.2.2. System Configurations

The MR switch device driver module was loaded into the FreeBSD kernel, integrated on top of the PCI bus enumeration modifications as detailed in Chapter 5.1.2. During boot-time, the OS probes all buses for PCI devices. For each discovered device, the OS finds the appropriate driver module with the highest priority level and attaches it to the PCI system structure. The bus is probed in a depth-first search manner, so once the MR switch device is probed and attached, the OS will proceed to probe and attach the devices downstream of the switch.

The driver itself is divided into two components, the user space component and the kernel space component. The user space implementation triggers the MR switch reconfiguration by passing the state of the desired partitioning to the device driver in kernel space via IOCTLs. A set of register and port operations, shown in Table 2 were implemented. The user space code, first uses register configuration operations to set the registers listed in Table 1 according to a desired partition configuration, and then calls the appropriate port configuration operation to create and/or update the kernel PCI data structures.

Table 2: Set of configuration operations.

Operation	Parameters
Register Configuration ▪ Access a device's PCIe CSR via ID routing	▪ Slot number ▪ Bus number ▪ Function number ▪ Register ▪ Number of Bytes ▪ Data value
Register Configuration ▪ Access a device's MMIO address space via address routing	▪ Address ▪ Register value
Port configuration ▪ Add port ▪ Remove port ▪ Resume port ▪ Suspend port	▪ Switch port number ▪ Bus number

5.2.3. Device Driver Operations

An IOCTL interface to register access was implemented, providing access to the PCI CSR and to the MMIO address spaces. ID routing to access the CSR was done through the standard FreeBSD interface `pci_cfgregread` and `pci_cfgregwrite` which reads and writes to a specific register. This provides the ability to modify the configuration of each active port in the PCI hierarchy.

Figure 5-4: Functions for register access a device's CSR.

```
u_int32_t pci_cfgregread (int bus, int slot, int func, int reg, int bytes)
void pci_cfgregwrite     (int bus, int slot, int func, int reg, u_int32_t data, int bytes)
```

Address routing to access the MMIO address spaces was done through the standard FreeBSD `bus_space_*` interface. These functions take as parameters the tag and handle on a resource, such as the Base Address Registers (BAR) associated with a memory mapped region, as well as an

address offset. The BARs are obtained from the PCI configuration space, and FreeBSD's mechanism for requesting resources from a parent bus returns the tag/handle for the BAR. Applying an offset to the BAR provides access to a specific device register mapped into virtual memory. Access to these registers is needed to modify the switch configuration registers, listed in *Table 1.*

Figure 5-5: Functions for register access to a device's MMIO space.

```
uint32_t bus_space_read_4 (bus_space_tag_t space, bus_space_handle_t handle,
                                                  bus_size_t offset)

void bus_space_write_4    (bus_space_tag_t space, bus_space_handle_t handle,
                                                  bus_size_t offset, uint32_t value)
```

Interfaces to several port operations were implemented and triggered via IOCTLs. These include the ability to add, remove, suspend or resume a port from a virtual switch partition.

Adding a port to a virtual partition is the most complicated configuration option. We assume that the enumeration process allocated sufficient space for new ports to be added to the switch hierarchy. At this point, if an I/O device is downstream of the added port, in order to properly attach the new device the driver must also scan the system for the availability of memory, I/O and interrupt resource space. *Figure 5-6* shows the process for adding a port to the MR switch downstream bus.

Figure 5-6: Pseudocode to add a port to the MR switch.

```
    parameters:
        - sc       = MR switch driver softc data structure
        - pcibus   = handle on the downstream bus of the switch
        - portnum  = device/slot number of the added child
        - busnum   = bus number assigned to new port
    return int = 0 if success, 1 if failure

1   int plxsw_bus_add_child(struct plxsw_softc * sc, device_t pcibus, int portnum, int busnum)
2       sc->secbus, sc->subbus ← Extend bridge [secbus:subbus] range to include busnum
3       struct pci_devinfo *dinfo newdinfo ← pci_read_device(device_t pcib, ...) to
4                                            initialize new pci_devinfo for child
5       device_t child ← pci_add_child(pcibus, newdinfo) to create new device_t
6       if allocate_resource_space(sc, pcibus, child, subbus) == True
7           bus_generic_attach(pcibus)
8           return 0;
9       else
10          return 1, port not added
```

Breaking down the pseudocode, Line 2 updates the secondary and subordinate bus numbers of the top level switch device to include the bus number of the new port (`busnum`). Line 3 initializes a FreeBSD defined `pci_devinfo` data structure for the child. Line 5 creates a new `device_t` data structure using the newly instantiated `pci_devinfo`, and adds it to the PCI hierarchy, downstream of `pcibus`. Line 6 calls the `allocate_resource_space` function, explained in detail in Chapter 5.2.4. This function scans the memory and I/O address space and determines if sufficient space exists, if so the new device may be attached via FreeBSD's `bus_generic_attach` function in Line 7, which probes the downstream ports of `pcibus` and attaches the new child.

Removal of a port from the downstream port of the MR switch was implemented according to the procedure shown in *Figure 5-7*. Line 2-3 retrieves a list of all children on the downstream port of the switch and proceeds to iterate through the list, searching for a child with a device/port number that matches the `portnum` input parameter, the number of the port to be removed. If the matching port is found, the secondary and subordinate bus numbers of the switch device are updated to exclude `busnum` in Line 4. In Line 5, FreeBSD's `device_delete_child` function removes the child and recursively detaches all children in its downstream hierarchy. Detaching a device frees the resources allocated to that device.

Figure 5-7: Pseudocode to remove a port from the MR switch.

```
   parameters:
       - sc       = MR switch driver softc data structure
       - pcibus   = handle on the downstream bus of the switch
       - portnum  = device/slot number of the child to be removed
       - busnum   = bus number of removed port
   return int = 0 if success, 1 if failure

1  int plxsw_bus_remove_child(struct plxsw_softc *sc, device_t pcibus, int portnum, int busnum)
2      for each child c of pcibus do
3          if (device number of c == portnum)
4              sc->secbus, sc->subbus ← Reduce bridge [secbus:subbus] range to exclude busnum
5              device_delete_child (pcibus, c)
6              return 0
7      return 1, port not removed
```

Through testing, it was determined that removing a port was not the safest of solutions. Cached data or resources of the port may continue to exist even after its removal, used by other threads or referenced in other system data structures. It is difficult to fully flush all cached sources. Interrupts also pose a problem. A device's interrupt stream must be stopped and its interrupt handler removed before it can be detached. This requires that all threads that may have executed this interrupt handler to return and release all mutexes. A more robust solution is to simply suspend the port which retains the memory space allocated to its downstream devices and if needed again, simply resume it with its resources intact. If memory resources are limited, then devices can be removed to recycle the space. On NetApp's filers, since resource space is plentiful, suspending the port is the safer alternative.

Figure 5-8: Pseudocode to resume a port from the MR Switch.

```
   parameters:
       - sc        = MR switch driver softc data structure
       - pcibus    = handle on the downstream bus of the switch
       - portnum   = device/slot number of the child to be resumed
       - busnum    = bus number of resumed port
   return int = 0 if success, 1 if failure

1  int plxsw_bus_resume_child(struct plxsw_softc *sc, device_t pcibus, int portnum, int busnum)
2      for each child c of pcibus do
3          if (slot number of c == portnum)
4              sc->secbus, sc->subbus ← Extend bridge [secbus:subbus] range to include busnum
5              return bus_generic_resume(c)
6      return 1, port not resumed
```

Figure 5-9: Pseudocode to suspend a port from the MR Switch.

```
   parameters:
       - sc        = MR switch driver softc data structure
       - pcibus    = handle on the downstream bus of the switch
       - portnum   = device/slot number of the child to be suspended
       - busnum    = bus number of suspended port
   return int = 0 if success, 1 if failure

1  int plxsw_bus_suspend_child(struct plxsw_softc *sc,device_t pcibus, int portnum, int subbus)
2      for each child c of pcibus do
3          if (slot number of c == portnum)
4              sc->secbus, sc->subbus ← Reduce bridge [secbus:subbus] range to exclude busnum
5              return bus_generic_suspend(c)
6      return 1, port not suspended
```

Figure 5-8 and *Figure 5-9* shows the pseudo code to resume and suspend a port respectively. Similar to the remove port function, Lines 2-3 iterate through the list of children on the downstream port,

and searches for the port to be resumed/suspended. If found, Line 4 updates `secbus` and `subbus` and Line 5 calls the FreeBSD functions `bus_generic_resume` and `bus_generic_suspend` respectively to configure the child.

5.2.4. Resource Allocation – Add Device

There are three types of system resources that need to be managed during MR switch reconfiguration: Interrupts, I/O decoders and Memory-Mapped I/O (MMIO) decoders. In a static system, resource assignment is typically handled by the OS at start-up. The OS probes for a new device, and determines its resource requirements by querying the two BARs of a Header Type 1 device and the six BARs of a Header Type 0 device. It then writes the starting address of allocated space to its CSR and attaches the device. If the device is a bridge, the OS also allocates resources to its child bus, proceeding in a depth-first search of all resource requirements downstream. But, when dynamically adding a device to the system, the driver must fully emulate this process. It must reconfigure the systems memory and address space accordingly, and doing so in a way that is transparent to the system user. It must first initialize the CSRs of a new device with the appropriate values such that the probe and attach process of the OS will proceed as it normally would.

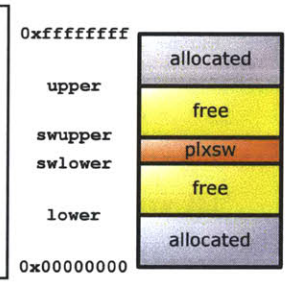
To determine these values, the driver must determine 1) the state of allocation of the existing system resources and 2) the resource requirements of the new port hierarchy. To add a port to the virtual partition, the driver must query the requirements of its downstream hierarchy. Whether or not the resources are implemented must be indicated in the Bridge Command Register in the port's PCI Configuration Space. The Command Register includes a flag for I/O Space enable at bit 0, a flag for Memory Space enable at bit 1 and a flag for Interrupt Disable at bit 10.

Figure 5-11 presents the driver implementation of the `allocate_resource_space` function, called from `plxsw_bus_add_child` (explained in *Figure 5-6*). This function scans the PCI hierarchy

storing the ranges of any I/O and MMIO address spaces in use, and then determines if the location and size of the available resource space is sufficient for allocation to the added device. If so, it appropriately sets and updates the registers in the PCI configuration space. A data structure, `addr_range` was defined and used to store temporary address ranges as the PCI bus was traversed. Its parameters are outlined in *Figure 5-10*.

Figure 5-10: Data structure storing range of allocated addresses.

```
struct addr_range{
    uint64_t lower      upper bound of allocated memory for devices with
                        memory ranges below that of the switch hierarchy
    uint64_t swlower    lower bound of allocated memory for switch
                         hiearchy
    uint64_t swupper    upper bound of allocated memory for switch
                         hierarchy
    uint64_t upper      lower bound of allocated memory for devices with
                        memroy ranges above that of the switch hierarchy
}
```



`Allocate_resource_space` initially calls the `scan_plxsw_branch` function in Line 4 followed by the `scan_pci_tree` function in Line 5. These functions respectively looks for MR switch resources, and non-MR switch resources. `Scan_plxsw_branch` traverses the MR switch hierarchy starting from the bridge, recursively searching the branch using depth-first search. For each device, any implemented BARs are decoded as MMIO or I/O addresses. When this function completes, the contiguous range of addresses mapped to the switch hierarchy is determined. The `scan_pci_tree` function, scans the entire PCI bus space by iterating through all `bus:device:function` combinations querying for devices not in the switch hierarchy, and when found, decodes any implemented BARs. When this function returns, the contiguous range of addresses mapped to the PCI hierarchy excluding the switch hierarchy is determined.

The traversal procedure takes advantage of the fact that as buses are enumerated in sequential order from 0 to 255, MMIO and I/O space is also allocated sequentially starting from `0x00000000` up to `0xffffffff`. Hence addresses allocated to bus numbers before the primary bus of the switch, will come before the addresses allocated to the switch, which in turn will come before addresses allocated

to the remainder of the devices in the PCI bus. In order to add a port to a switch partition, its mapped addresses must be adjacent to the address range of the switch hierarchy. Therefore, to successfully attach a port and its child devices, there must be sufficient address space between `lower` and `swlower`, or between `swupper` and `upper` to allocate to it (See *Figure 5-10* for illustration).

Figure 5-11: Pseudocode for resource scan and allocation.

```
    Parameters:
        - struct plxsw_softc *sc   Switch device softc
        - device_t pcibus          Handle on the downstream bus of the switch
        - device_t child           The new device to be added to switch
        - device_t busnum          The bus number of the new device
    Return: int = 0 if successfully set resources, 1 otherwise

1   int allocate_resource_space (struct plxsw_softc *sc, device_t pcibus, device_t child,
2                                int busnum)
3       struct addr_range memrange, iorange;
4       memrange, iorange ← scan_plxsw_branch(device_t plxsw_head)
5       memrange, iorange ← scan_pci_tree()
6       uint64_t mem_reqsize, io_reqsize ← get_resouce_reqsize(device _t child) +
7                                          get_resouce_reqsize(device_t *child->children)
8       if  memrange_exists(uint64_t mem_reqsize, addr_range memrange) &&
9           io_range_exists(uint64_t io_reqsize, addr_range iorange)
10              dev ← Set PCI config MEMBASE, MEMLIMIT, IOBASE, IORANGE registers
11              for each child of dev
12                  child ← Set config MEMBASE, MEMLIMIT, IOBASE, IORANGE, COMMAND registers
13              return 0
14      else
15          return 1, not enough resources
```

```
1  void scan_plxsw_branch (device_t dev, struct addr_range **memrange,
2                                        struct addr_range **iorange)
3      memrange, iorange ← get_device_bar( dev )
4      for each child of dev
5          get_plxsw_addr_range(child, memrange, iorange)
```

```
1  void scan_pci_tree (struct addr_range **memrange, struct addr_range **iorange)
2      for all bus numbers b from 0 to 255 do
3          for all device numbers s from 0 to 32 do
4              for all function numbers f from 0 to 7 do
5                  if dev ← domain=0, bus=b, device=s, function=f exists
6                      if not in switch branch
7                          memrange, iorange ← get_device_bar(dev)
```

Line 6 in `allocate_resource_space` determines the MMIO and/or I/O space required of the added port hierarchy. Lines 8-10 check to see if a sufficiently large gap exists, and if so, accordingly updates the Memory Base, Memory Limit, I/O Base and I/O Limit registers of the bridge, the added port and, lastly if they exist, their children. Once set, the FreeBSD `bus_generic_attach` function, called

from `plxsw_bus_add_child`, will be able to retrieve these derived register values, allocate the resources and attach the new port to the PCI tree. If sufficient space is not discovered, then the reconfiguration returns an error without attaching.

The third resource, interrupts is the least intrusive resource to allocate. PCIe devices implement Message Signal Interrupts (MSI), using a memory write packet to transmit interrupt vectors to the root complex host bridge device which in-turn interrupts the CPU (19). A system can support up to 2048 interrupts, and their assignment to new devices is not as heavily constrained as it is with memory or I/O address space allocation. As long as the maximum is not reached, the OS can readily allocate the required number of interrupts to the new device.

6. Experiment I: Re-configuration using Device Driver

In this chapter we test the device driver reconfiguration operations described in the previous section in various setup scenarios. Testing was divided into two principle stages. Chapter 6.1 performs reconfiguration on the switch without any I/O endpoints, while Chapter 6.2 performs reconfiguration with Intel Ethernet cards.

6.1. Re-configuration with no I/O endpoints

The first series of tests was a simple one, to test the validity of the implementation for each port operation, without accounting for the complications of address space management. The MR switch was setup with no inserted I/O devices, in other words, had no children attached to any of the ports. Therefore no MMIO, I/O space or interrupts needed to be allocated to successfully attach a port. The switch was reconfigured, moving a port/device from one virtual partition to another. This migrated port was then suspended, and then resumed, to test these driver operations.

Figure 6-1 shows the result of moving Device 21 from its original host, Host 2, to Host 1, then suspending it. *Figure 6-1A* shows the initial switch partitioning, with three ports assigned to each partition. *Figure 6-1B* shows the reconfigured switch partitioning. After reconfiguration, Host 2 was no longer able to detect the migrated port, Device 21, and it was also verified that it was unable to access the port's CSR's. On the other hand, the new host (Host 1) had full access. Following this, Device 21 was suspended with the resulting bus diagram shown in *Figure 6-1C*. The host was no longer able to detect or access the device in its PCI hierarchy. When the resume operation was called on Device 21 the bus returned to the state shown in *Figure 6-1B* and returned the suspended device to the hierarchy.

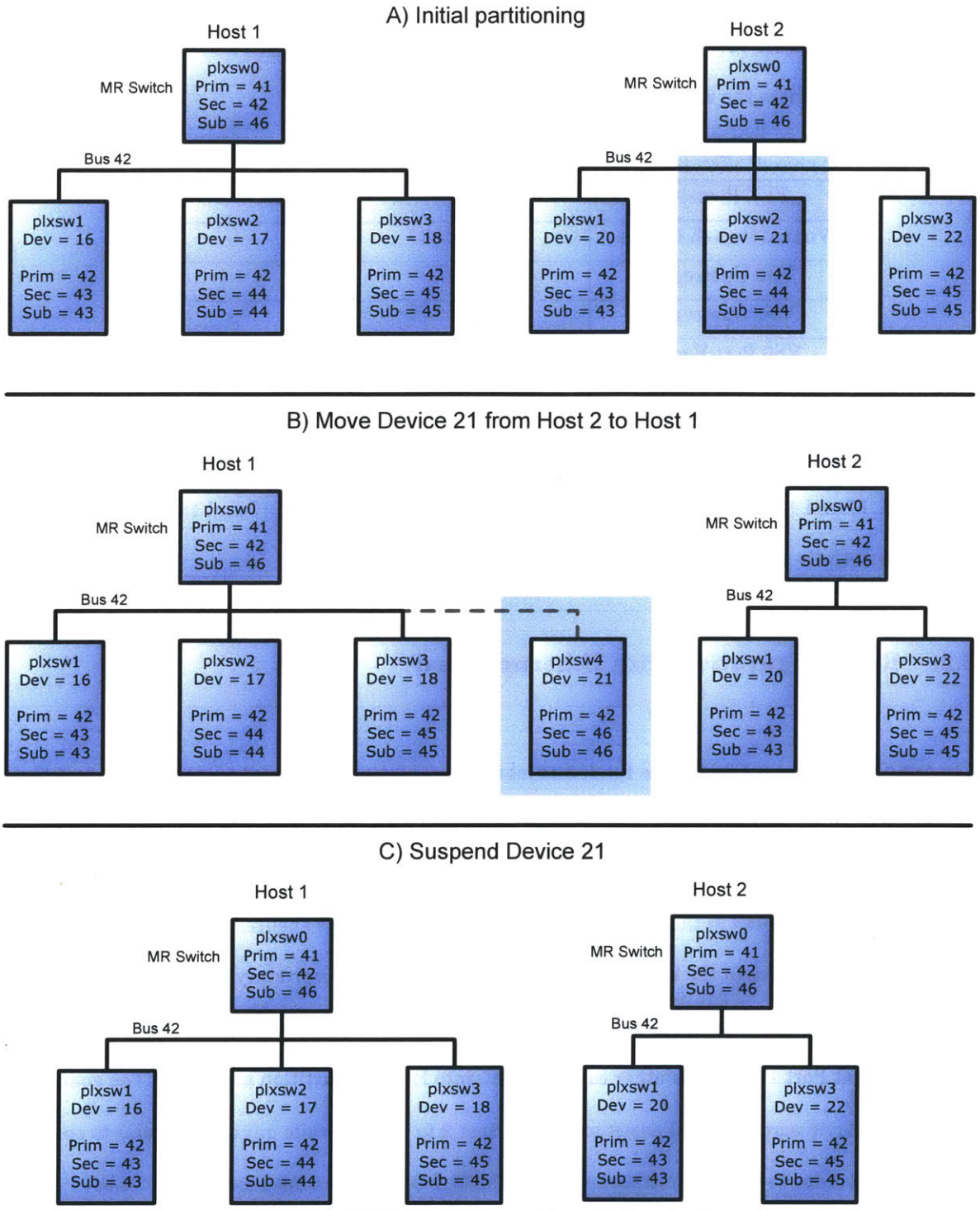


Figure 6-1: PCI bus from test with no I/O endpoints.

A) Bus from the initial switch partitioning. B) Bus from the reconfigured switch partitioning, moving Bus 42's Device 21 from Host 2 to Host 1's partition. C) Suspension of Device 21 removes it from the hierarchy.

Figure 6-2 shows the set of PLX configuration registers that initialized this specific partitioning. The two registers VS0 Port Vector and VS1 Port Vector were updated to include and exclude Port 21 respectively to initiate the reconfiguration process.

Figure 6-2: Updated PLX configuration register values.

PLX Register	Initial	Add Port 21
Virtual Switch Enable	0x00000003	
VS0 Upstream	0x00000000	
VS1 Upstream	0x00000004	
VS0 Port Vector	0x00070001	0x00270001
VS1 Port Vector	0x00700010	0x00500010

A series of additional tests were run: adding Devices 20, 21 and 22 from Host 2 to Host 1, removing Device 16, 17 and 18 from Host 1 and suspending/ resuming Devices 16, 17 and 18 on Host 1. All of which successfully updated the PCI bus.

6.2. Re-configuration with Ethernet Card

The tests in Chapter 6.1 tested the fundamental implementation of the port operations. To test the implementation of resource allocation, two Intel Ethernet card were used. These cards were used for two reasons: 1) they have relatively small resource space requirements, which provide a clean experiment scenario, and 2) their widespread use in today's systems ensures that default Intel drivers are pre-existing in the kernel and no additional driver or proprietary software is required to load it. The experiment was setup as follows: the first Ethernet card, `cardA`, was inserted into Host 1's partition at Port 17, and the other card, `cardB`, into Host 2's partition at Port 20. The goal was to migrate `cardB` across virtual partitions, from Host 2 to Host 1, while `cardA` continues to operate on Host 1. In order to attach an Ethernet card, MMIO, I/O and interrupt resources must be allocated. Errors allocating any of them will prevent a successful Ethernet connection on the host.

In *Figure 6-3* we note five instances of the switch driver, designated with name and unit number `plxsw[0...4]`. The newly attached port, `plxsw4`, is attached to the next available bus, Bus 46. There are also four instances of the Ethernet card driver, `em0` and `em1` are loaded for each function of `cardA`, and `em2` and `em3` for `cardB`. Ethernet cables were inserted into the jacks of `em1` and `em3` to establish two physical network connections.

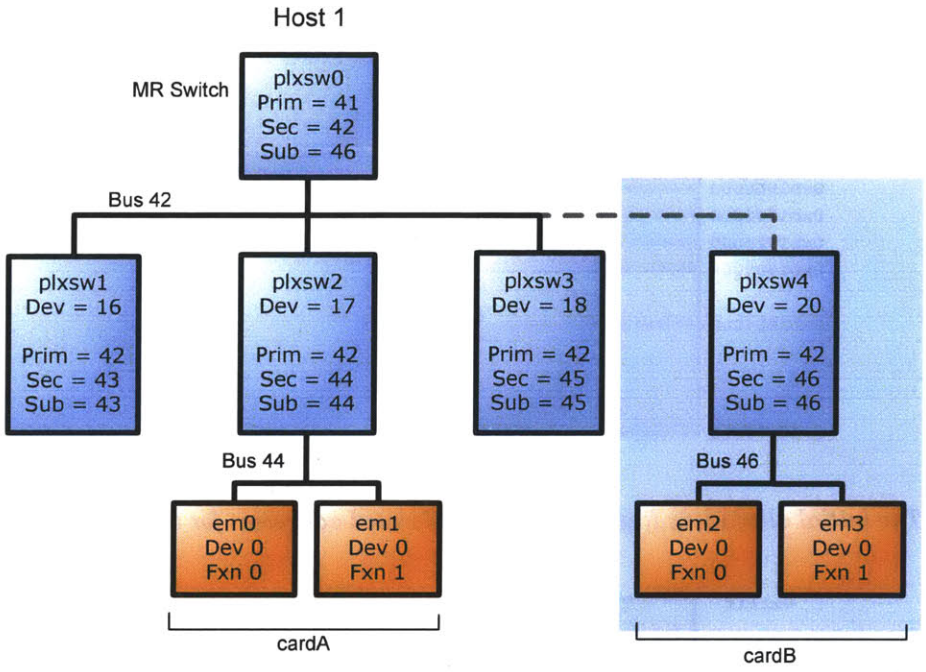


Figure 6-3: Host 1 PCI bus after Ethernet card migration from Host 2.

After probing the Ethernet device's CSR, it shows a memory space requirement of `0x80000h`, an I/O requirement of `0x20h` and two interrupt lines. From *Figure 6-3* we see on the left the initial address allocations in the memory stack, and on the right, the newly allocated addresses, assigned to the gap between `0xb1a40000` and `0xb1b00000`, initially owned by the root port. *Figure 6-4* shows the initial I/O stack on the left and the newly allocated addresses on the right, utilizing the gap between `0x1040` and `0x2000`.

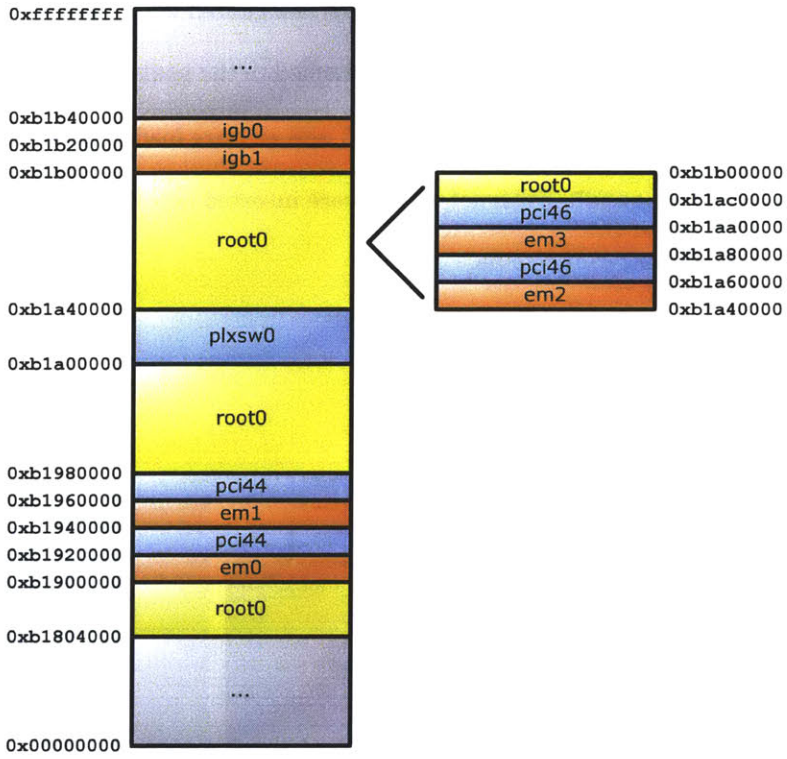


Figure 6-4: The memory stack.
Left: The initial allocation; Right: The newly allocated addresses for the added Ethernet card.

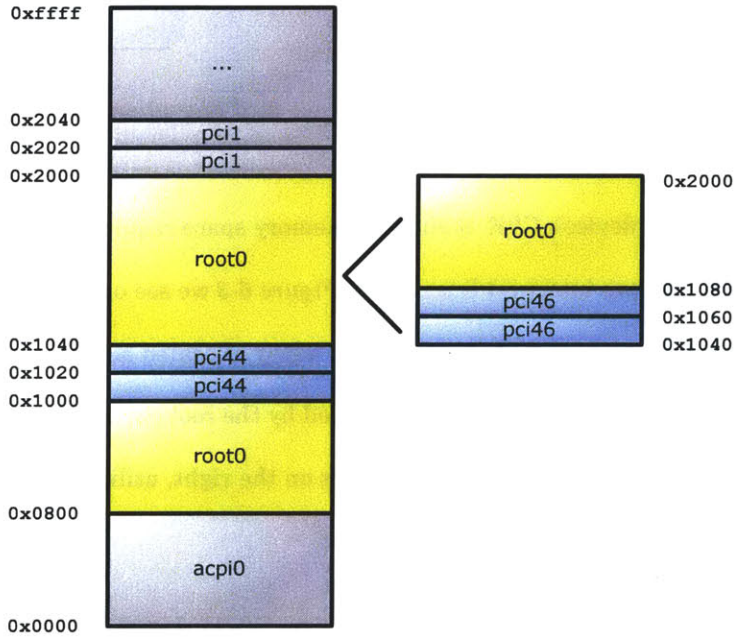


Figure 6-5: The I/O stack.
Left: The initial allocation; Right: The newly allocated addresses for the added Ethernet card.

Using these derived ranges, the limit and base register pairs for the bridge and children were written into the CSR. *Figure 6-6* shows the modifications made to CSRs of the switch bridges. *Figure 6-7* shows the modifications made to the CSRs of the endpoint Ethernet devices.

Modified register

Device Driver	plxsw0 (initial)	plxsw0 (new)	plxsw2	plxsw4
Header Type	1	1	1	1
Bus:Dev:Fxn (in hex)	29:00.0	29:00.0	2a:11:0	2a:14:0
Membase	0xb190	0xb190	0xb190	0xb1a0
Memlimit	0xb190	0xb1a0	0xb190	0xb1a0
IObase	0x11	0x11	0x11	0x11
IOlimit	0x11	0x11	0x11	0x11
BAR0	0xb1a00000	0xb1a00000	0xb1a00000	0xb1a00000

Figure 6-6: Updated registers in the switch configuration space.

Device Driver	em0	em1	em2	em3
Header Type	0	0	0	0
Bus:Dev:Fxn (in hex)	2c:00:0	2c:00:1	2e:00:0	2e:00:1
BAR0	0xb1900000	0xb1940000	0xb1a40000	0xb1a80000
BAR1			0xb1a60000	0xb1aa0000
BAR2	0xb1920000	0xb1960000		

Figure 6-7: Updated registers in the Ethernet device configuration space.

In order to verify the successful attachment of the migrated Ethernet card, we need to verify its full functionality in the new system. A network throughput benchmark, Netio was used to do this. Netio measures the throughput via TCP/IP with varying packet sizes. Netio compared the performance of `em3` on `cardB` with the performance of the experiment constant,`em1`, on `cardA` and also to the performance of the built in Ethernet card on the host machine, `igb0`. *Figure 6-8* and *Figure 6-9* graph the measured transmitting (Tx) and receiving (Rx) bandwidths. It reveals comparable

performance between all three Ethernet devices, suggesting that there are no performance repercussions to dynamically inserting an Ethernet card through switch reconfiguration.

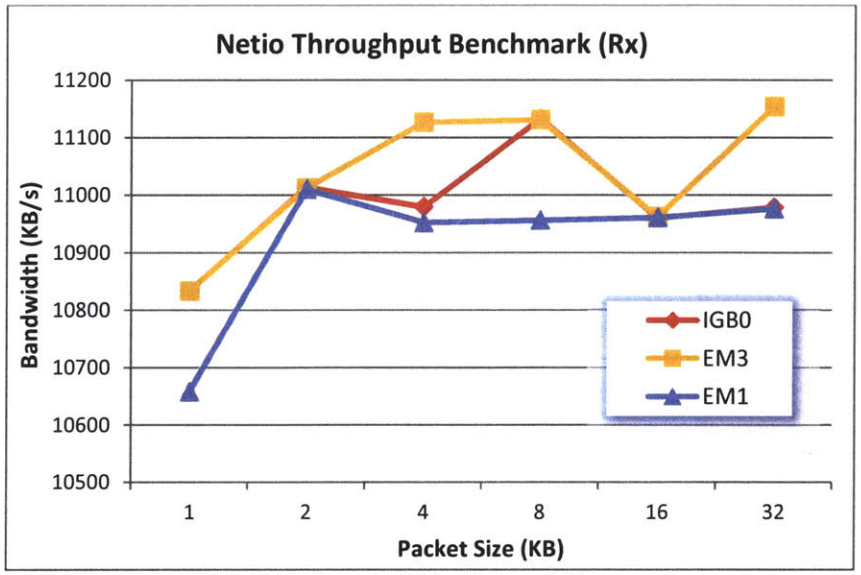


Figure 6-8: Receiving bandwidth of three Ethernet connections.

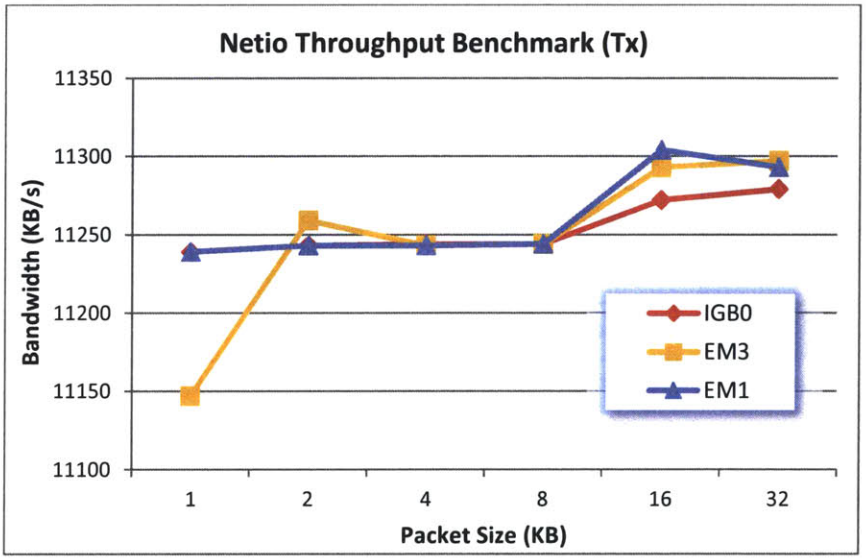


Figure 6-9: Transmitting bandwidth of three Ethernet connections.

7. Experiment II: MR Switch Performance Validation

The previous chapters demonstrated the feasibility of implementing dynamic system reconfiguration using a MR switch. In this section we evaluate the performance of our MR switch prototype. Our goal is to verify that there is no significant performance degradation when using a system with a MR switch versus one without. One such consideration is the effect on latency. Appending a switch to the hierarchical path from host to the endpoint device increases the path length which may in turn increase the latency of each I/O transaction.

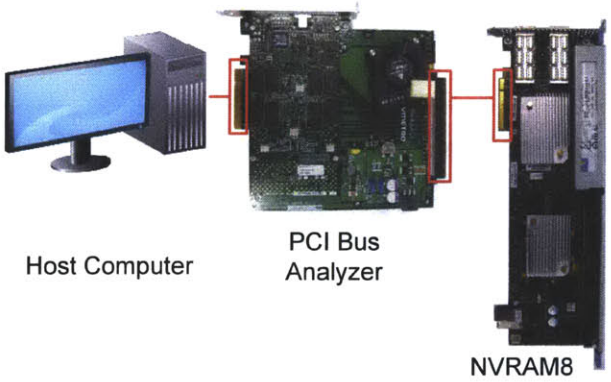


Figure 7-1: Setup to measure latency through host system.

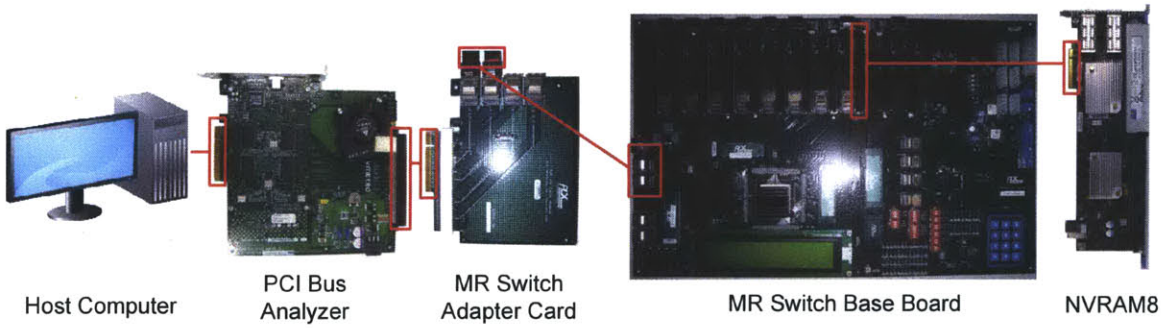


Figure 7-2: Setup to measure latency through the MR Switch.

The specifications for the PEX 8664 indicate a maximum packet latency of 176ns with a x16 to x16 connection between symmetric ingress and egress ports. In order to validate that the latency of an I/O operation is not significantly greater than this performance guideline, we run two performance

tests. Both tests involve using NetApp's proprietary NVRAM8 DMA engine to generate packet traffic through the MR switch. The traffic was measured using two software tools: 1) an internal NetApp benchmarking tool tailored for the NVRAM8, and 2) Vanguard's VMetro PCI bus analyzer, a third party development package. In the tests we compare the latency of a transaction in a system with the NVRAM8 plugged directly into the host computer (*Figure 7-1*) versus the latency of having it inserted into a MR switch downstream port (*Figure 7-2*).

Figure 7-3 compares the recorded latency measurements using both measurement tools where VG denotes Vanguard Bus Analyzer, and NTAP denotes the NetApp performance tool. Host denotes the test setup in *Figure 7-1* and VS denotes the test setup in *Figure 7-2*.

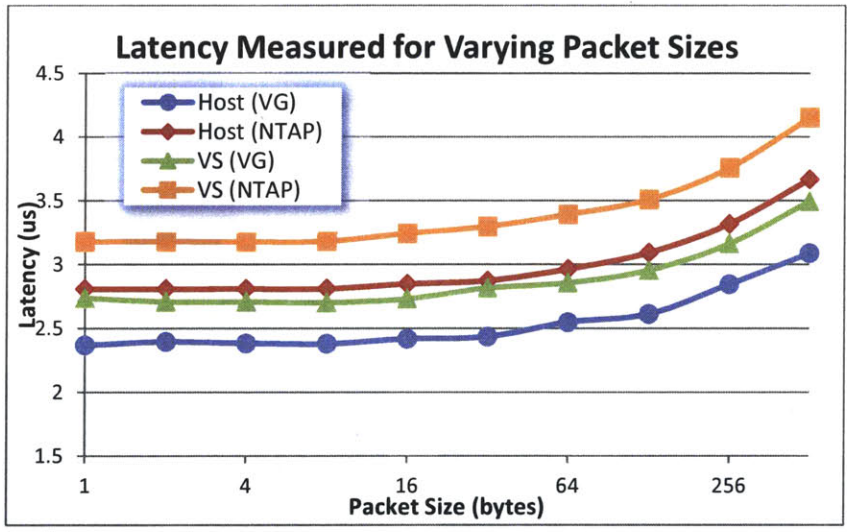


Figure 7-3: Graphs the measured latencies from each test.

8. Discussion

In this chapter, we analyze the results from the re-configuration experiments and performance tests. We look at it successes, its limitations and also suggestions for improvements.

8.1. Re-configuration Process

We presented two test environments for reconfiguration, and in both cases, demonstrated that the MR switch reconfiguration process behaved according to expectation. Adding, removing, suspending and resuming a MR switch port with no I/O endpoints attached, and also with an Ethernet card attached worked properly, with the changes in the switch configuration accurately reflected in the host PCI bus data structures. In the more interesting case, dealing with the Ethernet card, its success demonstrates that the procedure and approach to resource allocation in the device driver implementation is sound. Furthermore, it shows that dynamic system reconfiguration is feasible using the systems and resources readily available in today's industrial environment. But the success of the experiment itself does not fairly represent the magnitude of improvement in flexibility, reliability or scalability that can be achieved through the MR switch.

The experimental platform in this thesis had many factors that limited the scope of the re-configurability of the switch. Namely, the complexity of the reconfiguration was limited by the available resource space (MMIO and I/O space) in the functionally-basic host systems. As mentioned previously, the Ethernet card was selected for its generality, and small resource requirements, which were satisfied by the resource capabilities of these particular hosts. But if more complex I/O endpoints are used, these systems will not provide a plausible test ecosystem.

In a more specialized industrial environment, such as NetApp's, more powerful and configurable systems are available. NetApp's filers have significantly large resource spaces, and most

importantly, these spaces are fully configurable. By having full control over MMIO and I/O address allocation, resource spaces could be partitioned, with significant gaps reserved adjacent to the MR switch space to be assigned to reconfigured devices. The host systems used in the experiments were equipped with 32-bit addressing, referencing a 4GB memory space. But in systems equipped with 64-bit addressing a much larger address space is available, allowing for more flexibility when partitioning this space. These platform modifications must be taken into account when analyzing the full extent of the potential of integrating a MR switch.

8.2. Performance

From the four sets of tests shown in *Figure 7-3*, we see latencies varying between 2.36µs and 4.15µs reflecting the time it takes for the packet to travel upstream and downstream. As expected we see a greater latency with the addition of the MR switch. We also note that the measurements of NetApp's internal tool returns values slightly higher than that of the Vanguard analyzer. This is also expected, since NetApp's tools are used for internal validation and early phase testing, and not fine-tuned to produce highly accurate results. *Figure 8-1* graphs the latency increase with the inserted MR switch. We see an average degradation of approximately 407ns using NetApp's tool, and 340ns using Vanguard's tool, which translates to a one way latency of 204ns and 170ns respectively. Both these numbers are well within the expected range of the 176ns latency stated in the switch specifications.

These results show that the effects of adding a MR switch into the packet path from root to endpoint is considerably small and easily accommodated for. Though a more thorough analysis of the impact of introducing a 170ns latency into the transaction path may be needed in more complex platform scenarios.

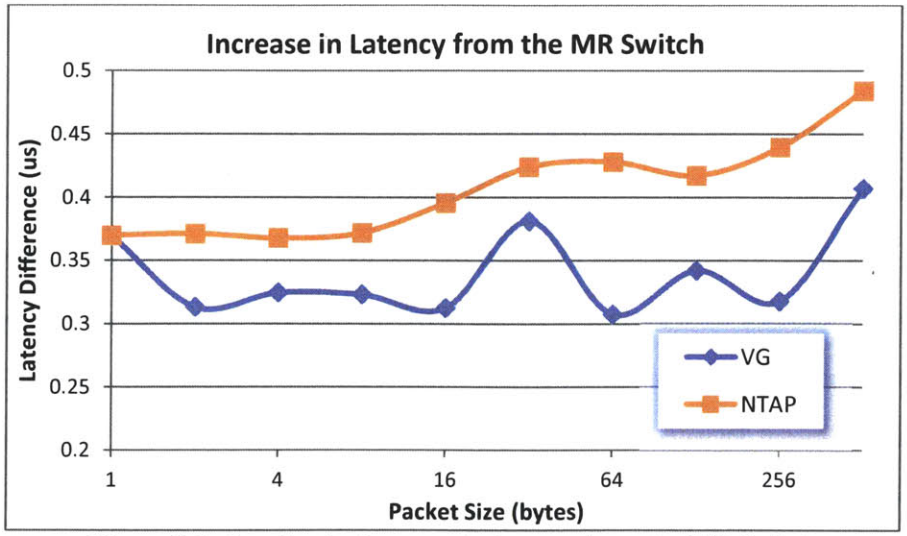


Figure 8-1: Graphs the increase in latency from the MR Switch.

An alternative performance consideration is the time it takes for the entire reconfiguration process to run. For the series of experiments run in Chapter 6, reconfiguration took a short amount of time, within a time frame of several milliseconds. Though it must be noted that this measurement is fairly baseless, as it will depend on the driver implementation, but to a must larger extend the hardware and software specifications of the host system and the testing ecosystem. Regardless, reconfiguration time is not of serious concern since reconfiguration is not likely to occur very frequently. A slightly increased runtime will have not a huge impact.

9. Conclusion

In this thesis, a methodology has been developed to integrate a switching solution using a PCIe multi-root switch during system operation on a FreeBSD platform. Using the device driver written for the PLX PEX 8664, we successfully executed the adding, removing, suspending and resuming operations on a series of switch ports in a testing platform with two host systems. The experimental results demonstrated that changes in the switch configuration were accurately reflected in the host PCI bus data structures. From this we can conclude that reconfiguration of the MR switch in a live system is very feasible and implementable in today's systems.

There are many features that can be explored in the future to make data centers and bladed server systems more efficient. One such application is to use the multi-root switch to provide I/O device failover at the platform level to increase system availability and reliability. In today's data centers, failover architecture is typically integrated at a higher, more software oriented level. It is rarely a feature offered at the platform level, supported by the infrastructure. But if dynamic switch reconfiguration is implemented, it would be possible to migrate the workload of a failed host across virtual partitions to an alternate functional host, allowing the failed host to be removed, repaired and then replaced. After which, the configuration of the switch can be reversed to return the workload to its initial host. All this will of course be transparent to the users. To do this, support for failover detection must be implemented. This can be done by polling the PCIe Status configuration register, which indicates the status of the link. If the status of an upstream port shows a severed link, the connection to the host may be lost, and in turn should trigger the process of failover switch reconfiguration.

A second application is enhancing resource management and load balancing between multiple root complexes. Leveraging the configurability provided by the multi-root switch, it would be possible to better manage resource usage of a cluster. Workload can be migrated from overworked systems to

underutilized ones, moving unused I/O devices across partitions to systems that would more greatly benefit from it.

The PCI Express multi-root switch partitionable architecture provides system architects with an unprecedented level of flexibility. These prototype switches, developed over the past several years has the potential to offer a solution to many of the needs of today's data center platforms. The ability to dynamically reconfigure the allocation of I/O devices to multiple hosts or root complexes in a live system has great merit, and with the maturing of the switch hardware, the software needs to quickly follow suit.

10. Bibliography

1. **Jones, Matt, IDT.** PCIe switch enables multi-host systems. s.l. : EE Times-India, 2009.
2. **Jones, Matt.** Optimizing PCI Express Switch and Bridge Architectures. s.l. : RTC Magazine, May 2005.
3. **Bhatt, Ajay.** *Creating a PCI Express Interconnect.* 2002.
4. **Ravi Budruk, Don Anderson, Tom Shanley.** PCI Express System Architecture. Boston, MA : MindShare inc., 2004, pp. 9-54.
5. **Regula, Jack.** Using Non-Transparent Bridging in PCI. [Online] 6 1, 2004. [Cited: 03 10, 2011.] http://www.eetimes.com/design/embedded/4006788/Using-PCIe-in-a-variety-of-multiprocessor-system-configurations.
6. **Regula, Jack.** Using PCIe in a variety of multiprocessor system configurations. [Online] 1 23, 2007. [Cited: 03 10, 2011.] http://www.eetimes.com/design/embedded/4006788/Using-PCIe-in-a-variety-of-multiprocessor-system-configurations.
7. **PCI-SIG.** *Multi-Root I/O Virtualization and Sharing Specification Revision 1.0.* 2008.
8. **Schrader, Bernhard.** Multi Root I/O Virtualization.. and its Potential to consolidate I/O Infrastructures. Paderborn, Germany : Fujitsu Siemens Computers, 2008.
9. **Jones, Matt.** *Enabling multihost system architectures with PCI Express switches: Innovation in design through multiroot partitionable switch architecture.* 11 18, 2010.
10. **Shanley, Tom and Anderson, Don.** PCI System Architecture 4th Edition. s.l. : MindShare Inc., 1999, p. 499.
11. **PCI-SIG.** *PCI Hot-Plug Specification.* 06 20, 2001.
12. **Sourceforge.net.** Linux Hotplugging. *About Hotplugging.* [Online] [Cited: 03 21, 2011.] http://linux-hotplug.sourceforge.net/.
13. **FreeBSD.** FreeBSD - PCIHotplug. [Online] 06 17, 2008. [Cited: 03 21, 2011.] http://wiki.freebsd.org/PCIHotplug.
14. **Intel Corp.** *Intel 5520 and Intel 5500 Chipset Rev 2.2.* 2010.
15. **PLX Technology Inc.** Express Lane PEX 8664-AA Data Book v1.3. San Jose : PLX Technology Inc., 2010, p. 245.
16. **IDT.** IDT 89HPES48H12G2 PCI Express Switch User Manual. San Jose : s.n., 2010, pp. 15-1.
17. **Intel Corp.** *Intel 82571EB Gigabit Ethernet Controller.* USA : Intel PRO Network Connections, 2005.
18. **eTesters.** VMETRO provides free host exerciser utility for PCIe, PCI-X and PCI. [Online] [Cited: 03 23, 2011.] http://www.etesters.com/news/story.cfm/itemID/145.
19. MSI-HOWTO.txt. [Online] http://www.mjmwired.net/kernel/Documentation/MSI-HOWTO.txt.
20. **PCI-SIG.** *PCI Standard Hot-Plug Controller and Subsystem Specification Rev 1.0.* 06 20, 2001.
21. **Huang, Derek and Kulkarni, Upendra.** *Multi-Port Spread Spectrum Clocking Support in IDT PCIe Gen2 Switches.* San Jose : IDT Inc., 2009.
22. **PLX Technology Inc.** *Choosing PCI Express Packet Payload Size.* s.l. : PLX, 2007.
23. **PCI-SIG.** *PCI Local Bus Specification Revision 3.0.* 2004.

24. **McKusick, Marshall Kirk, et al., et al.** *The Design and Implementation of the 4.4BSD Operating System.* s.l. : Addison-Wesley Longman, Inc., 1996.

25. **FreeBSD Foundation.** *FreeBSD Developers' Handbook.* s.l. : The FreeBSD Documentation Project, 2000.

26. **Tanaka, Hiroyyuki, Nomura, Yoshinari and Taniguchi, Hideo.** *Run-time Updating of Network Device Drivers.* s.l. : International Conference on Network-Based Information Systems, 2009.

27. **PCI-SIG.** *Single Root I/O Virtualization and Sharing Specification Revision 1.0.* 2007.

28. *Multi Root I/O Virtualization.* **Schrader, Bernhard.** Paderborn, Germany : Fujitsu Siemens Computers, 2008.

29. **Foundation, FreeBSD.** *FreeBSD Porter's Handbook.* s.l. : The FreeBSD Documentation Project, 2000.

30. **Dong, Yaozu, Yu, Zhao and Rose, Greg.** SR-IOV Networking in Xen: Architecture, Design and Implementation. Berkeley, CA : WIOV'08 Proceedings of the First conference on I/O virtualization, 2008.

31. **PCI-SIG.** PCI Express 1.1 Protocols & Software. s.l. : PCI-SIG Developers Conference APAC Tour, 2005.

32. **Regula, Jack.** Use PCIe in multi-processor system configurations. s.l. : PLX Technology, Inc., 2007.

33. **Nguyen, Tom Long; Sy, Dely L; Carbonari, Steven; Intel Corporation.** PCI Express Port Bus Driver Support for Linux. Ottawa, Ontario : Proceedings of the Linux Symposium, 2005.

34. **Kong, Kwok.** Using PCI Express as the Primary System Interconnect in multiroot Compute, Storage, Communications and Embedded Systems. s.l. : IDT, 2008.

35. **Kung, Shawn.** Storage Resiliency for the Data Center. s.l. : Network Appliance, Inc., 2006.

36. **Shah, Shreyas.** PCI Express and IOV: Maximizing Multi-Processor Systems. s.l. : PCI-SIG Developers Conference, 2008.

37. **FreeBSD Foundation.** *FreeBSD Architecture Handbook.* s.l. : The FreeBSD Documentation Project, 2000.

38. **Budruk, Ravi.** PCI Express Basics. s.l. : PCI-SIG, 2007.

39. **PLX Technology Inc.** Multi-Host System and Intelligent I/O Design with PCI Express. Sunnyvale, CA : s.n., April 2005.

40. **Pohlmann, Frank.** Why FreeBSD. [Online] IBM developerWorks, 07 19, 2005. [Cited: 03 22, 2011.] http://www.ibm.com/developerworks/opensource/library/os-freebsd/.