



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# Transaction Level Modeling of a PCI Express Root Complex

**Even Låte**

Electronics System Design and Innovation

Submission date: June 2014

Supervisor: Snorre Aunet, IET

Norwegian University of Science and Technology  
Department of Electronics and Telecommunications



# Project Assignment

**Candidate name:** Even Låte

**Assignment title:** Transaction Level Modelling of a PCI Express Root Complex

**Assignment text:**

Model a Peripheral Component Interconnect(PCIe) Root Complex (RC) with respect to important performance metrics such as latency, bandwidth and jitter.

An RC serves as a bridge between PCI and QuickPath Interconnect(QPI) for Intel architectures.

PCIe is a packet-based, serial interconnect standard that is widely deployed in workstation for it's attractive bandwidth capabilities. QPI is also a packet-based communication standard that is used internally in the RC between the "I/O hub" and Intel's "Unicore".

I/O-traffic is on most modern systems said to be I/O coherent, that is, I/O transactions work on concert wit the cache coherency of the CPU-memory interconnect. PCIe has the ability to avoid the coherency(NoSnoop), to relax the ordering between transactions(RO). PCIe also has provisions for placing/read-ing data directly from the host caches, through TLP processing hints(TPH), which is formerly known as direct cache transfer. All these, can if used correctly, in-crease the performance of the communication architecture.

The main focus of the assignment is to create a high level model of an RC and CPU/memory complex in order to provide performance simulation of a PCIe device in a relevant environment, with respect to latency, bandwidth and jitter. The model can be correlated with real hardware, using a PCIe tracer to observe the characteristics of traffic of traffic on a real hardware platform. This in combination to selected traffic scenarios is well suited to correlate the model.

<b>Assignment Proposer:</b>	Håkon Bugge, ORACLE
<b>Responsible Professor:</b>	Snorre Aunet, NTNU
<b>Supervisor:</b>	Morten Schanke, ORACLE



# Abstract

PCI Express(PCIe) is a packet-based, serial, interconnect standard that is widely deployed within servers and workstations for its attractive performance capabilities. A platform that has a PCIe architecture also includes a PCIe Root Complex(RC) for linking the PCIe device-tree to the host CPU and memory.

During the design-phase of a PCIe endpoint-device it is highly desired to conduct computer aided simulations of the device in a relevant environment. Having a simulation software that can be applied early and iteratively in the design-phase enables engineers to tweak the product without realization of hardware. Causing a great reduction in the number of physical prototypes required before mass production.

In this thesis a transaction level model(TLM) of a PCIe RC was assembled using SystemC, with a focus on latency and jitter as performance parameters. The model gives the Application Specific Integrated Circuit(ASIC) developers at Oracle a timing accurate alternative to the existing processor emulator(QEMU) that is used for the same purpose. To correlate the RC TLM with real hardware, a PCIe protocol analyzer from LeCroy was utilized. Traffic between a first generation PCIe endpoint-device and a SUN FIRE X4170 M3 server was traced.

The RC TLM was made in a modular manner allowing support for other micro-architectures through insertions of trace files. The recorded traces between requests and completions were processed and inserted directly into a delay database within the RC model, to ensure high correlation between the RC TLM and the real hardware. A simple model of a PCIe endpoint-device was implemented to serve as a suitable test-environment.

The functionality and the hardware realisticness of the RC model was successfully tested with targeted transaction scenarios. A simulated latency distribution of 15000 packets, proved to fit the latency distribution that was randomly drawn in the RC TLM. Only a small amount of negligible delay anomalies from imperative switch cycles were found.

The PCIe RC TLM is close to optimal for modeling latency and jitter using a database of targeted trace calibrations. The principle of modeling delays in an RC TLM using latency databases, was found to be a favorable alternative to the constant delay nature of the QEMU test-environment.



# Preface

This thesis is submitted to the Department of Electronics and Telecommunications at NTNU to complete a Master of Science degree in electronics; digital circuit design.

The assignment proposal originated through a meeting with Oracle in late November, 2013. The work on the thesis was initiated the 20th of January 2014, and was concluded the 16th of June the same year.

The intention of this assignment is to contribute to Oracle's existing ASIC test-environment by creating a transaction level model of a PCIe root complex for more hardware realistic performance simulations.

I would like to express my gratitude to; Morten Schanke and Håkon Bugge from the Oracle Corporation and professor Snorre Aunet from NTNU, for their support throughout these 21 weeks. I would also like to thank Oracle and its employees at Skullerud for including me in their work towards designing ASICs for the new generation of enterprise data-centers.

Trondheim, 2014-06-16

Even Låte





# Contents

<b>Project Assignment</b>	<b>i</b>
<b>Preface</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Thesis Overview . . . . .	3
<b>2 Theoretical Background</b>	<b>5</b>
2.1 PCI Express . . . . .	6
2.1.1 The PCIe Evolution . . . . .	6
2.1.2 The PCIe System Architecture . . . . .	9
2.1.3 The PCIe Topology . . . . .	22
2.1.4 PCIe Packet Sending Example . . . . .	26
2.2 PCIe Related Performance Metrics . . . . .	28
2.2.1 Bit Rate / Bandwidth . . . . .	28
2.2.2 Latency and Jitter/Packet Delay Variation(PDV) . . . . .	29
2.3 Root Complex Emulation with QEMU . . . . .	33
2.4 SystemC . . . . .	35
2.5 The LeCroy PCIe Gen. 1 Trace System . . . . .	38
<b>3 Methodology</b>	<b>41</b>
3.1 Decision of Approach . . . . .	42
3.1.1 Deciding on the Architectural Structure of the PCIe RC Model	42
3.1.2 Model Approach for Delay Correctness . . . . .	43
3.2 Implementation of the RC model . . . . .	44
3.2.1 Implementing the TLP C++ class . . . . .	44
3.2.2 Implementing the PCIe Encoder/Decoder Socket Module . . . . .	47
3.2.3 Implementing the PcieSwitch Module . . . . .	48
3.2.4 Implementing the Subsystem Memory Module . . . . .	50
3.2.5 Implementing the RC Module. . . . .	51
3.3 Performing traces on relevant HardWare . . . . .	54
3.3.1 Setting up the Trace Hardware and the Trace Software . . . . .	54
3.3.2 Performing the Tracing . . . . .	55
3.3.3 Converting the Trace Data to Text Delay Coloumns . . . . .	56

3.3.4	Linking the Delay Model and the Trace Data . . . . .	57
3.4	Creating a test-environment for the RC . . . . .	59
3.4.1	Constructing the PCIe endpoint model . . . . .	59
3.4.2	Interconnecting modules creating a complete system . . . . .	60
3.4.3	Creating the Program Interface in sc_main.cpp . . . . .	61
3.5	Testing the PCIe RC Model . . . . .	62
3.5.1	System functionality . . . . .	62
3.5.2	Valgrind Memory Leak Check for Runability . . . . .	71
<b>4</b>	<b>Results</b>	<b>73</b>
4.1	Packet Tracing . . . . .	73
4.1.1	The PETracer Recordings . . . . .	73
4.1.2	The Generated Delay File for the RC Model . . . . .	74
4.2	Functionality Test-Results of the RC . . . . .	77
4.2.1	Functional Accuracy . . . . .	77
4.2.2	Delay Model Accuracy . . . . .	92
<b>5</b>	<b>Discussion</b>	<b>95</b>
5.1	Analyzing the PCIe Trace Recordings . . . . .	95
5.1.1	The Recorded MRd-CpID Latency Distribution . . . . .	96
5.1.2	The Delta Delay Extraction tool . . . . .	96
5.2	Analyzing the RC TLM . . . . .	97
5.2.1	Implementing the RC TLM . . . . .	97
5.2.2	Testing The Functionality . . . . .	101
5.3	Using the RC model for performance testing of EPs . . . . .	105
5.3.1	Deciding on the PCIe performance criteria . . . . .	106
5.3.2	Implementing a TLM of the EP . . . . .	106
5.3.3	Running the Performance Simulations . . . . .	107
5.3.4	Evaluation of the RC TLM as a Tool for Performance Testing . . . . .	108
5.4	Future Work . . . . .	109
<b>6</b>	<b>Conclusion</b>	<b>111</b>
	<b>References</b>	<b>115</b>
<b>A</b>	<b>Acronyms</b>	<b>119</b>
<b>B</b>	<b>C++ Tool for Converting Exported Trace Files</b>	<b>121</b>
<b>C</b>	<b>Python Script for Plotting of Latency Distributions</b>	<b>125</b>

<b>D PCIe Traffic Trace Summaries</b>	<b>127</b>
D.1 Summary of Trace Iteration 1 . . . . .	128
D.2 Summary of Trace Iteration 2 . . . . .	132
D.3 Summary of Trace Iteration 3 . . . . .	136
<b>E Simulation Output Format</b>	<b>141</b>
E.1 Root Complex Completes a MRd Request . . . . .	141
<b>F Valgrind Test for Program Runability</b>	<b>145</b>



# List of Figures

1.1	MSI Z77 requiring Intel Ivy Bridge for PCIe gen 3 support[38]	1
2.1	Overview of the theory sections in this thesis	5
2.2	The PCIe logo by the PCI-SIG [26]	6
2.3	The evolution of the PCI and PCIe generations from 1992 until now	7
2.4	Two PCIe devices communicating with TLPs and DLLPs	9
2.5	The TLP and it's layer-targeted segments	10
2.6	The generic TLP header	10
2.7	The memory request header 3DW & 4DW	12
2.8	The completion header	13
2.9	The structure of the data link layer packet	13
2.10	The layered structure of the PCIe protocol stack	15
2.11	The software layer of the PCIe Protocol	15
2.12	The transaction layer of the PCIe Protocol	15
2.13	Type 1 and type 0 configuration register headers[3]	17
2.14	The data link layer of the PCIe Protocol	19
2.15	The physical layer of the PCIe Protocol	20
2.16	Topology of a typical PCIe system	22
2.17	The PCIe multiport RC	23
2.18	The PCIe Endpoint	25
2.19	A packet switch with VCB and Port arbitration	26
2.20	Example of a memory read transaction on a PCIe topology	27
2.21	MRd-CplD delays, decomposed	30
2.22	Jitter demonstrated, jitter equal to zero vs jitter equal to 1	30
2.23	Packet arrival time, normally distributed	31
2.24	Packet arrival time, heavy-tailed distributed	32
2.25	Weighted mixture of probability distributions	33
2.26	Intel's I440fx supports PCI and utilize ISA, available in QEMU	34
2.27	Intel's ich9 supports PCIe, now available in QEMU	34
2.28	A SystemC D-FlipFlop example	35
2.29	Trace hardware from LeCroy and CATC[20]	38
2.30	PETracer software snippet[20]	39
2.31	A hardware setup for the protocol analyzer	39

3.1	Outline of the project workflow . . . . .	41
3.2	Structural overview of the RC TLM . . . . .	42
3.3	A realistic delay model . . . . .	43
3.4	Modules, classes and functions of the RC TLM . . . . .	44
3.5	A class diagram of the TLP and its header structure . . . . .	45
3.6	A module diagram of the PCIe protocol stack . . . . .	47
3.7	A module diagram of the PCIe packet switch . . . . .	48
3.8	A module diagram of the host memory . . . . .	50
3.9	A module diagram of the RC TLM . . . . .	51
3.10	Trace setup at Oracle's test lab . . . . .	54
3.11	OFED comand for executing an application on the PCIe EP . . . . .	56
3.12	Snippet of raw packet flow between the PCIe EP and the RC . . . . .	56
3.13	Snippet of TLP-only flow between the PCIe EP and the RC . . . . .	57
3.14	Example of TLP Traffic extracted to Text . . . . .	58
3.15	The Text Format Required by the RC Delay Model . . . . .	58
3.16	A module diagram of the PCIe EP TLM . . . . .	59
3.17	A module diagram of the complete test system . . . . .	60
3.18	The command-line syntax for executing a simulation . . . . .	62
3.19	Testing a single MRd from EP:002 to the RC . . . . .	64
3.20	Testing a single MWr from EP:002 to the RC's system memory . . . . .	64
3.21	Testing a single MRd from EP:002 to the RC, RCB exceeded . . . . .	65
3.22	Testing 2 simultaneous requests, switching cycles are illustrated . . . . .	65
3.23	Testing the virtual channel buffer arbitration . . . . .	66
3.24	Testing rapidly sent requests . . . . .	66
3.25	Testing rapidly sent requests, RC relaxed ordering . . . . .	67
3.26	Testing RC's ability to issue requests, in this case a simple MRd . . . . .	68
3.27	Testing RC's ability to issue a simple MWr request . . . . .	68
3.28	Testing the RC's ability renew lost requests . . . . .	69
3.29	Testing the maximum payload size restriction with a memory write . . . . .	69
3.30	Testing bidirectional MRd traffic . . . . .	70
3.31	Testing bidirectional MWr traffic . . . . .	70
3.32	A realistic PCIe scenario, computation is outsourced to the EP . . . . .	71
3.33	The command-line syntax for a Valgrind execution . . . . .	71
4.1	Overview of the result sections in this chapter . . . . .	73
4.2	A histogram of the total amount of traced delays . . . . .	74
4.3	The first sub distribution in figure 4.2, heavy-tailed . . . . .	75
4.4	Fitting a probability distribution to the first sub distribution . . . . .	75
4.5	The last sub distribution in figure 4.2, bell shape . . . . .	76
4.6	Fitting a probability distribution to the last sub distribution . . . . .	76
4.7	The summary of the output.txt file generated by a single simulated MRd request from EP 2:0:0 . . . . .	78

4.8	The summary of the output.txt file generated by a single simulated MWr request from EP 2:0:0 . . . . .	79
4.9	The summary of a large MRd request, proving the RCB functionality . . . . .	80
4.10	Simulation log file that illustrates the imperative delta delay consumed by switch modules . . . . .	81
4.11	Simulation log file that illustrates the packet priority of the system, EP 4:0:0 is handled first because of the delta delay in the switch . . . . .	82
4.12	Simulation log file from a simulation with multiple MRd requests sent with low periodicity, illustrates clumping in the RC . . . . .	83
4.13	Simulation log file from a simulation with multiple MRd requests sent rapidly, multitasking in the RC resolves the clumping . . . . .	84
4.14	Simulation log of the Root Complex using address routing to route a single MRd package downstream to its reciever . . . . .	85
4.15	Simulation log of the Root Complex using address routing to route a single MWr package downstream to its reciever . . . . .	86
4.16	Simulation log of the Root Complex using address routing to route a single MRd package downstream to its reciever . . . . .	87
4.17	Simulation log of the Root Complex sending a single MWr request for a large amount of data, being limited by the max payload attribute . . . . .	88
4.18	Simulation log of a simulation where all devices are requesters and potentially completers . . . . .	89
4.19	Simulation log of a simulation where all devices requests memory writes to the other devices . . . . .	90
4.20	Simulation log of a realistic PCIe system scenario . . . . .	91
4.21	Simulation log of 15000 packets over 1 ms . . . . .	92
4.22	Correlation between sampled and simulated latency distributions . . . . .	93
5.1	Topics of discussion and their sections in this chapter . . . . .	95
5.2	Mixture probability distribution function of the MRd-CplD delays . . . . .	98
5.3	A blue print of a generic EP TLM to ensure RC TLM compatibility . . . . .	107
F.1	Summary log for a valgrind run, no definitely lost memory . . . . .	145





# List of Tables

2.1	Paralell bus structures, frequencies, transfer rates and io ports [3]	7
2.2	The PCIe Transaction types . . . . .	9
2.3	The generic header fields of the TLP, explained . . . . .	11
2.4	The memory-request header specific fields, explained . . . . .	12
2.5	The completion-header specific fields, explained . . . . .	14
3.1	The settings used for a trace recording . . . . .	55
4.1	TLP-type trace statistics summary for all 3 trace iterations . . . . .	74
4.2	General statistics for the recorded traces . . . . .	74



# Introduction

## 1.1 Motivation



**Figure 1.1:** MSI Z77 requiring Intel Ivy Bridge for PCIe gen 3 support[38]

Today's industry shows an increasing trend towards large data-centers and databases. Internet traffic is expected to increase exponentially as the introduction of Internet Protocol version 6 (IPv6) allows the Internet of Everything (IoE) to be created. Cisco estimated that around 50 billion devices will be connected to the Internet in 2020 as a result of IPv6 and the IoE [7]. All these devices generate massive amounts of data that requires processing, communication and storage. Organizations manage more data, with greater speed and from more sources than ever before.

Larger IT-based companies desire to ease the job of the average IT-technician by acquiring hardware and software solutions that are able to virtually tie machines together. They want to fuse many sparse databases into few virtual ones that are easily manageable. This demand creates a market for virtual, secure and high performance database solutions. Oracle's ASIC-department at Skullerud in Oslo works to satisfy the need for quicker transfer of data between devices. They

are part of Oracle's global R&D team with a focus on communication between the internal processing units of server systems. The ASICs utilize communication protocols such as PCI Express(PCIe) and Infiniband to move data from one processing unit to another due to their attractive transfer rates.

PCIe is a packet-based, serial, interconnect standard. It is widely deployed and has a bright future as a communication infrastructure for peripheral units in computers. Figure 1.1 shows a MSI Z77 motherboard, incorporating a third generation PCIe slot. PCIe devices communicate with other devices, in a switch based tree structure. A common PCIe topology contains a root complex(RC), several Endpoints(EPs) and optional switches. The RC is the logic structure that interconnects the CPU, the memory and the PCIe Tree containing the system I/O. The EPs are I/O devices that are connected to the branches of the PCIe tree, an example would typically be a graphics controller. The switches allow data transactions between devices by applying packet switching on behalf of the RC or EPs in the PCIe tree.

It is crucial to map system performance when developing PCIe hardware, both during the design process, and afterwards for marketing purposes. Teledyne Lecroy and Agilent deliver solutions for protocol analyzation of PCIe Hardware. Physical test instruments are able to interpose PCIe Traffic and to record detailed packet information that can be used to analyze performance of realized PCIe EPs. However, it is also desired to simulate device performance during the design-phase of a PCIe device, prior to hardware manufacturing. Simulations allow tweaking of hardware in front of realization, which is valuable with respect to design-cost and performance due to a reduction of prototype-iterations. Computer aided simulations are easily configured and allows for optimization of parameters such as buffer sizes, algorithms and other system variables. The result is that latency, jitter and packet congestion are reduced, maximizing the bandwidth and the quality of service(QoS) of the final product.

To perform software simulations of PCIe express EPs, a model of the RC, and models of the EPs are required. Oracle's ASIC-branch in Oslo and in California have until now been using QEMU for environment modeling during hardware simulations. QEMU is a generic open source machine emulator and virtualizer, that includes an emulated CPU with an RC and a memory system. Some problems with QEMU as a RC simulator is that it is not cycle accurate, propagation delays are not included in round-time latencies and there is no model for jitter included in the round-time latencies. Another factor is that it contains a lot of excess functionality that is not necessary from a PCIe-ASIC developer's point of view. There is a need for a new tool that fills these gaps.

This thesis involves the creation of an accurate and purpose specific alterna-

tive to the already existing PCIe simulation environment QEMU. A transaction level model(TLM) of a PCIe RC is to be created. Accuracy of the model should be ensured by correlating it to real PCIe hardware through protocol analyzations of a relevant server platform.

## 1.2 Thesis Overview

The successive chapters delve into and analyze the creation of a transaction layer model of a PCIe RC. Each chapter is marked with a black rectangle in the upper right corner of the page. On the very same page an illustration is given describing the contents of the chapter's child sections. An overview of the paper, chapter by chapter, is given below:

Chapter 2 provides background information on theory regarding terms and concepts that will come in handy throughout the paper. The chapter begins with a detailed overview of PCI Express protocol. In-depth knowledge of PCIe is required to understand the nature of a PCIe RC. PCIe related performance metrics are also explained. Theoretical performance-limits of a PCIe system are seldom achieved, the actual performance of the system is dependent on a wide specter of system variables. Terms such as bit-rate, net bit-rate, peak bit-rate, bandwidth, throughput and latency are elucidated. The chapter then moves on to describing the QEMU software. Basic knowledge of the transaction-level modeling language, SystemC, is summarized to ease the understanding of the implementation. The benefits of using SystemC compared to other modeling languages are also explained. The Final section of the theoretical background chapter covers PCIe protocol analyzing. Both software and hardware components of the trace instruments are introduced to draw a picture of the packet tracing process.

Chapter 3 covers the methodology of this thesis, that is basically the workflow for the entire design process of designing an RC TLM. The first section covers the decision phase of the structural architecture of the RC model. The following section describe the selection of a modeling language for implementation of the model. The workflow of the software implementation is described, including the design of internal modules, classes and functions used within the TLM. This chapter also describes the process of obtaining traffic information from recording traffic on real, first generation PCIe hardware. The data achieved from the protocol analyzer is merged with the RC's delay model to create a realistic delay model of the root complex. The latter sections of chapter 3 presents the implementation of a compatible TLM of a PCIe EP, test and verification of the RC model are also outlined.

Chapter 4 presents illustrations of the results from the PCIe traffic record-

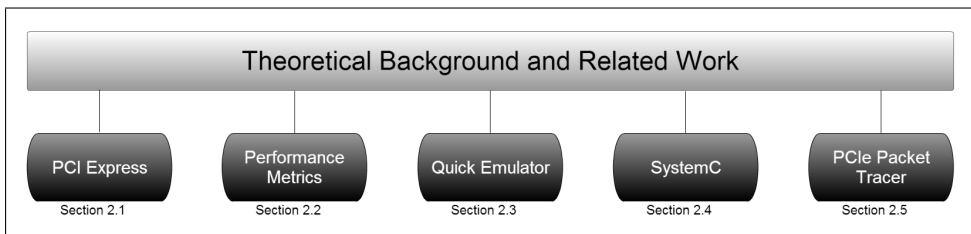
ings from the LeCroy protocol analyzer. The remaining section of this chapter will list the test-results achieved from testing the software model, both with respect to functionality and timing accuracy. The result chapter will consist of several windows containing exported text-files from test-runs of the model. It could be advantageous to read this chapter electronically as text out-puts are more easily interpreted with search functions.

Chapter 5 brings out the discussion of the results. The created software model of the PCIe Root complex is analyzed to map whether the model is an adequate option to simulation using QEMU. Requirements for use are also discussed, these are; compatible implementation of an EP TLM in SystemC and instantiation of all the components in the complete simulation environment. The chapter covers several topics, all attempting to answer the question: Is the RC TLM's functionality better than that of the QEMU emulator with respect to hardware realismness? Topics that are covered are: functional adequacy, simplifications of the software implementation, future work of the model and challenges faced during implementation.

Chapter 6 is the conclusion chapter, it contains answers to the questions discussed throughout this thesis. Will the work carried out in this thesis benefit Oracle in the form of a platform that has the potential to reduce development costs and improve the final performance of their PCIe ASICs?

Acronyms used in this thesis are listed in appendix **??**. For readers who find the implementation of an RC TLM interesting, the appendixes can provide essential details for the software made for this assignment. Some basic examples of program output from the RC system-model are attached in the appendix. The c++ script for converting exported files and the python script for plotting latency distributions are to be found. The trace files them selves are not included as they are in the gigabyte size-range, however summaries of them are attached. The Code itself for the RC TLM, can be found on Git-Hub at the location specified in [10].

# Theoretical Background



**Figure 2.1:** Overview of the theory sections in this thesis

The purpose of this chapter is to introduce several important concepts and terms that are used in the modeling approach of the RC and in the discussions throughout this thesis. Figure 2.1 shows the outline of this chapter. The PCI Express section is the most weighted one in this chapter, this section gives a detailed overview of the transaction layer nature of the PCIe Architecture. PCI Express relevant performance metrics are also discussed in this chapter to reason for the modeling approach of the Root Complex. A section is dedicated to explain the Quick EMUlator software currently used by Oracle. The event driven nature of the SystemC modeling language is covered in its own section to ease reading of the model and to ease the understanding of the diagrams found in the methodology chapter. Finally, the hardware that is used for recording of PCI Express traffic is clarified to ensure a hardware-realistic TLM model of the PCI Express Root Complex.

## 2.1 PCI Express



**Figure 2.2:** The PCIe logo by the PCI-SIG [26]

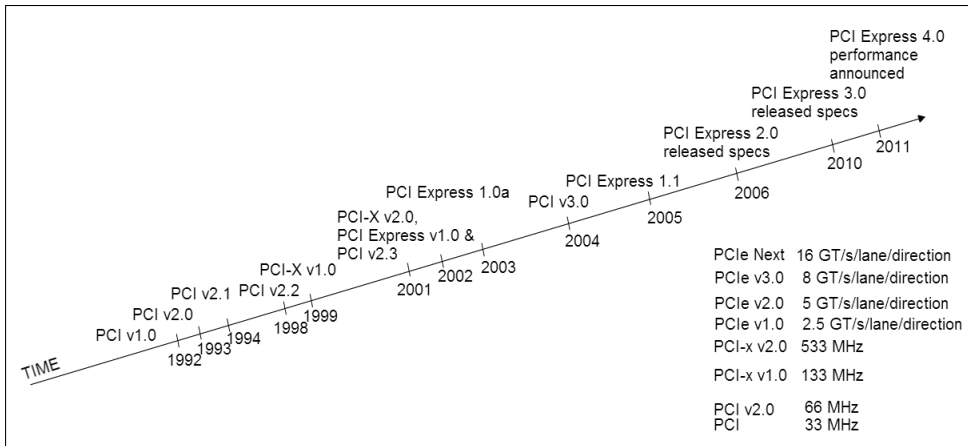
The Peripheral Component Interconnect Express standard is officially abbreviated PCIe. It is a serialized, point to point, connection-standard developed by the Peripheral Component Interconnect-Special Interest Group (PCI-SIG). It is a packet-based interconnect that allows peripherals to communicate with other devices by the use of differential signaling. The peripherals that are interconnected in a third generation PCIe system are achieving transfer rates of up to 8(GT/s) billion transfers per second, per lane, per direction. This corresponds to a raw bit rate total of 16(GB/s) billion bytes per second, per direction, for a 16-lane PCIe slot. These terms and metrics are described in detail later in this chapter.

PCIe is applied in technologies such as personal computers, servers, embedded computing and communication platforms. The most commonly known field of application for PCIe architectures is as interconnections between CPUs and graphics cards within personal computers. The system host writes data to the memory that is adjacent to the graphic accelerator. Once the data transfer is complete, the host issues a kernel call to provide the graphics processing unit(GPU) with data processing instructions. A completed outsourced computation is written back to the host once it is complete. Whenever graphics cards desire to fetch data from CPU-adjacent memory, or write data to it, they have to consult with the PCIe controller first, also known as the Root Complex(RC).

### 2.1.1 The PCIe Evolution

In the early computer-days around 1970, before the old PCI standard was invented, data was transferred using serial connections. Computers assembled data-packets and sent them serially from device to device, one packet at a time. As time went by, serial connections proved to be slow despite their reliable and robust nature. To cope with the steep speedup of computers, scientists began to develop parallel structures for communication, oblivious to the fact that they





**Figure 2.3:** The evolution of the PCI and PCIe generations from 1992 until now

were stepping into a design-architecture with less potential for performance. This increasing trend gave birth to the PCI bus which was introduced 1992 as shown in the time line in figure 2.3. For years the 32 bit PCI bus was the most convenient manner to connect sound, video and network cards to a motherboard.

Bus Type	Clock Frequency	Peak Bandwidth	#Card Slots
PCI 32-bit	33 MHz	133 MBytes/sec	4-5
PCI 32-bit	66 MHz	266 MBytes/sec	1-2
PCI-X 32-bit	66 MHz	266 MBytes/sec	4
PCI-X 32-bit	133MHz	533 MBytes/sec	1-2
PCI-X 32-bit	266 MHz effective	1066 MBytes/sec	1
PCI-X 32-bit	266 MHz effective	2131 MBytes/sec	1

**Table 2.1:** Paralell bus structures, frequencies, transfer rates and io ports [3]

Incremental improvements of bus transfer speeds were achieved by increasing bus clock frequencies. The PCI theoretically support 32 devices connected to each bus. Increasing the bus frequency in parallel bus architectures turned out to drastically reduce the number of feasible electrical loads. This is because the method for detecting bus devices rely on the reflection of a driven test signal to locate the termination of the bus. The more devices connected to the bus, the longer it takes for the signal to be reflected. The total time it takes from the signal is set to the reflection arrives must be less than the clock period to for the signal to reach a device before bus evaluation.

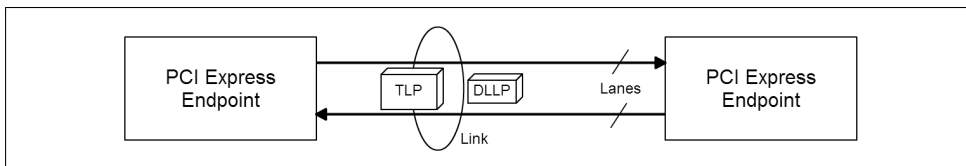
Higher requirements to data-transfer rates together with the electrical load barrier was why PCI-X was introduced. The multi-drop parallel bus began to replace the PCI, however it was only a temporarily solution as computers grew more and more powerful. Table 2.1 reflects this phenomenon, increasing the

frequency for a given architecture implies reducing the number of card slots per bus, both for PCI and PCI-X. The electrical load barrier created yet again a need for a new, revolutionary design approach. Computer scientists were forced to re-think fundamental concepts as they were facing a physical barrier.

Research advancements in the field of dividing, labeling and re-assembling packets as well as improvements in electrical hardware components for differential signaling, re-introduced the trend of serial data transfers. Interconnect research had again turned towards high speed serial communication which gave birth to PCIe, a serialized packet-based communication protocol. PCIe 1.0 was introduced in 2002 as a third generation I/O bus. Today, high speed interconnects such as USB and Firewire and PCIe are all built on the serial data transfer principle. PCIe kept the strong features of PCI, and improved the weak ones, the serial transfer approach, makes it technically speaking, not a bus. The predecessors of PCIe are referred to as true buses, they are constructed with physically adjacent rails for parallel data transfer. PCIe on the other hand side, has a structure that resembles a serial network connection. It mimics PCI even though it communicates using a packet based protocol. The resemblance to PCI ease the task of migrating PCI devices to PCIe, create bridges between the two, and removes the need to change software during a migration.

The PCIe bus is said to be categorized as a third generation bus out of three generations:

- The first generation of buses consists of multiple purpose specific bus-standards, all with dedicated tasks to be assembled into a larger system. One bus for is dedicated for memory, one for peripherals and so. Many smaller systems are built together to create a larger computer system. Examples of first generation buses are: ISA, EISA, VESA and Micro channel buses.
- The second generation of buses are known for using two layers of connection. The CPU and the memory are connected to the north-bridge, and the peripheral devices are connected to the south-bridge, the two modules are interconnected with an internal bus structure. Examples of second generation bus standards are: PCI, AGP, PCI-X.
- Third generation buses are flexible with respect to the number of physical connected devices to the system. This makes them suitable for use internally in computer systems connecting computing nodes together. The third generation of buses behave like networks rather than bus standards.



**Figure 2.4:** Two PCIe devices communicating with TLPs and DLLPs

## 2.1.2 The PCIe System Architecture

### PCIe Units of Transaction

This section covers the system architecture of PCIe, starting from the bottom up with the units of transactions first, then the device layers and finally the coarse grained system topology. PCIe is as mentioned above a packet based communication protocol. There are two packet types flowing on the links between PCIe devices. These are the transaction layer packets(TLPs) and the data link layer packets(DLLPs) shown in figure 2.4. The DLLPs are used for Link management functions, for transferring data between devices that help maintain connection statuses and to provide information about reception of TLPs. Examples of such functions are ACK/NAK handshakes, exchange of flow control details and power management features. The TLPs are the larger communication units that are used for the transactions themselves. That includes memory actions, io actions, configuration of PCI devices and for messaging and event reporting. The TLPs are explained in detail below as these are essential for the creation of an RC TLM.

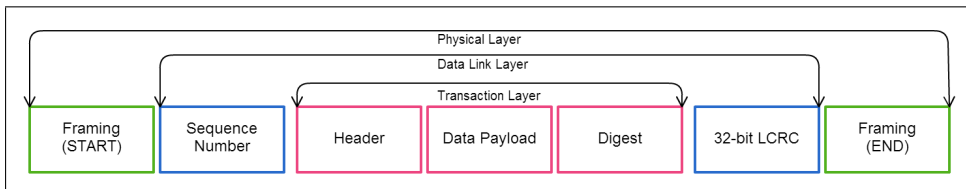
### The Transaction Layer Packet

The 15 transaction layer PCIe transactions are listed in table 2.2 in 5 categories.

Category	TLP Types	Acronym	Posted or Non-Posted
Memory	Memory Read Request	MRd	Non-Posted
	Memory Read Lock Request	MRdLk	Non-Posted
	Memory Write Request	MWr	Posted
IO	IO Read Request	IORd	Non-Posted
	IO Write Request	IOWr	Non-Posted
Configuration	Config Type 0 Read Request	CfgRd0	Non-Posted
	Config Type 0 Write Request	CfgWr0	Non-Posted
	Config Type 1 Read Request	CfgRd1	Non-Posted
	Config Type 1 Write Request	CfgWr1	Non-Posted
Message	Message Request	Msg	Posted
	Message Request W/Data	MsgD	Posted
Completion	Completion	Cpl	-
	Completion W/Data	CpID	-
	Completion-Locked	CplLk	-
	Completion-Locked W/Data	CplDLk	-

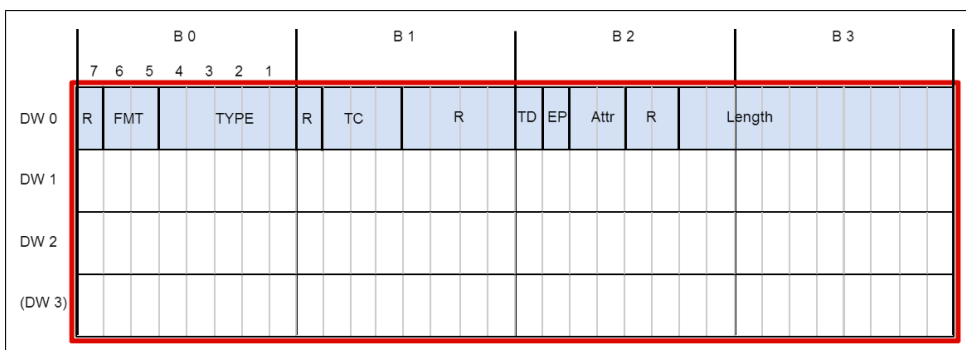
**Table 2.2:** The PCIe Transaction types

The requests can be split into two types, posted transactions and non-posted transactions. The non-posted transactions are those that expect TLP answers in return of a request, typically memory read requests that expect a completion TLP containing the requested data. The posted transactions are those that do not expect a reply such as memory write requests. Writes are satisfied once the data is written to the specified system memory address. Non-Posted requests are handled as a split transactions, freeing the bus while waiting for an answer. This is an automatic feature as a result of the packet based nature of PCIe.



**Figure 2.5:** The TLP and its layer-targeted segments

The TLP is consists of the 7 main segments shown in figure 2.5. The header of the TLP is assembled in the transaction layer of the sender, and disassembled in the transaction layer of the receiver. The transaction layer section of the TLP is encapsulated with sequence numbers, error checks and start-stop indications on its way down towards the egress port. These encapsulations are upon reception of the packet, peeled off like onion-shells in their respective abstraction layers. When the packet arrives at the software layer of its destination, only the payload and the header remains.



**Figure 2.6:** The generic TLP header

The header segment of the TLP shown in figure 2.5, consists of 3 or 4 Double Words(DW). The data payload is up to 1024 DW data payload and the optionally ECRC segment is 1 DW . The 15 various types of TLPs listed in table 2.2 are distinguished from each-other by differences in the byte-fields within the header of the packet. The structure of the header varies for the different TLP types, which is represented with the white fields in figure 2.6. The light blue fields are

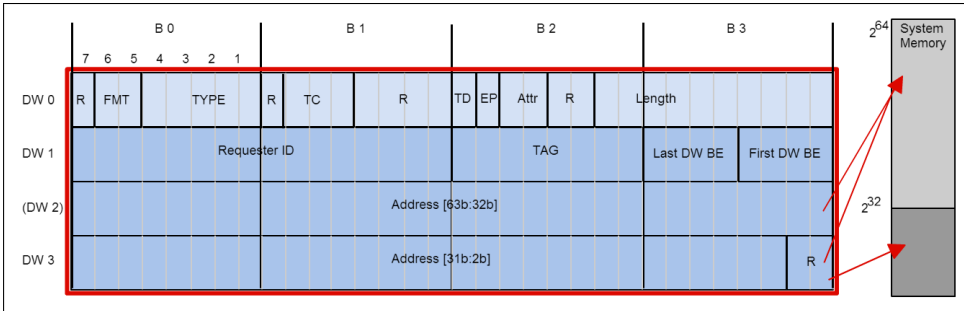
Header field	Explanation
Reserved (R)	The reserved fields are all reserved for header recognition, they are all set to 0's during TLP construction. Recent patches to the PCIe structure use some of the reserved fields for extra attributes such as extended tags, and TLP hints [18] [40].
Format (Fmt)	The format of the TLPFmt are used to tell whether a data payload is present in the TLP or not, it is also used for information about the length of the header itself, if it is 3 DWs or if it is 4 DWs.
Traffic Class(TC)	The Traffic class is the priority of the TLP, it tells what priority based virtual channel buffer it is to be put in at forwarding and handling. The higher the number the higher is the priority, TC can be a number from 0 to 7.
TLP Digest (TD)	TLP digest tells whether the 1DW optional ECRC section of the TLP is present or not, the receiver of the packet must check the ECRC field if it is.
Poisoned Data (EP)	If this bit is high, the TLP is considered to be invalid. The transaction is allowed to complete normally.
Attributes (Attr)	The attributes of the header tells whether to <b>Relax Ordering (RO)</b> or to use strict ordering when transferring the packet. The attributes also tell whether to set <b>No Snooping (NS)</b> for the transaction, meaning that no cache coherency issues exist regarding this TLP. The system is allowed to skip snooping to ensure cache coherency thus increasing the system performance.
Length	The Length field indicates the size of the data payload in given in DW, maximum payload size is 1024DW as the field is 10 bits long.

**Table 2.3:** The generic header fields of the TLP, explained

generic, they are common for nearly all packet types. Each generic type field is explained further in detail in table 2.3.

The header types for memory requests packets and completion packets are explained in this paper as these are vital for an RC TLM. The type specific fields for MRd and MWr are shown in figure 2.7. Equally, Cpl and CplD header fields are described in figure 2.8. Explanations for the memory request packets are found in table 2.4 and completion header explanations are given in table 2.5. The gray rectangle next to the memory request header represents 32bit addressing vs 64 bit addressing in system memory, the 3 dw memory request header can only access the darker shade of gray, the bottom 4 GB of PCIe system memory. Please refer to the PCI e system manual [3] for explanation of the remaining header types for the packets listed in 2.2

To further understand the packet structures and the intention behind the packet types, a deeper look into the layered structure of the PCIe protocol is necessary as the DLLPs originate in the data link layer, and the TLPs originate in the transaction layer of PCIe devices.



**Figure 2.7:** The memory request header 3DW & 4DW

Memory Header field	Explanation
Last DW Byte Enable	Last DW Byte Enables consist of four bits that indicate what bytes that are included in the last DW of the payload, this field is needed as the address field is DW aligned.
First DW Byte Enable Requester ID	First DW Byte Enables works in a similar fashion to Last DW BE, it tells what bytes to include in the first DW of the data payload. The requester ID tells the completer of the memory request who the requester is, so that a completion packet may be returned. The requester ID is the device ID of the requester, described by a bus number from 0 to 255, a device number from 0 to 31, and a function number from 0 to 7. Completion packets for memory requests are returned using ID routing
Tag	The Tag of the memory request is a water-stamp that is put on a non-posted request packet to ID outstanding requests that are issued by a requester. By default a device may only have 32 outbound requests at a time. The completion packet to the request is also water-stamped with the same tag, so that the request can be removed from the outbound buffer in the transaction layer of the receiver. Recent changes to this parameter is explained in [40], the change is called extended tags, allowing 256 outbound requests.
Address	The address field of the memory request header provides the system memory address for the requested memory action. The entire PCIe system is given different segments of the 64 bit available system memory address space, this allows for address routing to be used when routing packets between PCIe devices. The address field is either 1 DW or 2 DWs, depending on the addressing type indicated in the format of the header. If the header is 3 DW in total, only the lower 4 GB might be addressed. The lowest 4 bits of the address forces the memory to be double word aligned, creating the need for the byte enable fields.

**Table 2.4:** The memory-request header specific fields, explained

### The Data Link Layer Packet

The DLLPs originate in the data link layer of a transmitter and are terminated in the data link layer of a receiver. The DLLPs are used for management of the PCIe

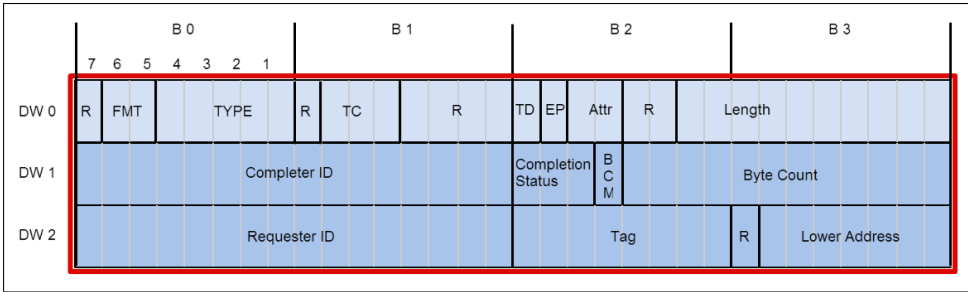


Figure 2.8: The completion header

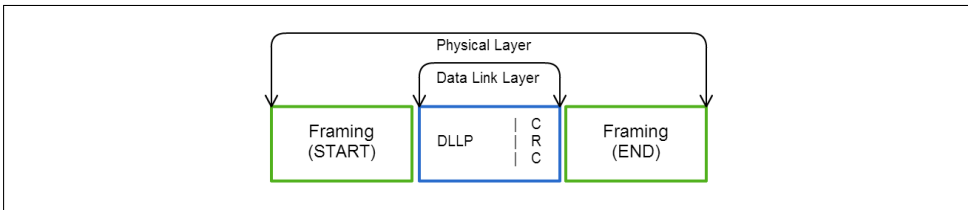


Figure 2.9: The structure of the data link layer packet

links by sending ACK/NAK packets, power management information, and to exchange credits that are used for flow control of virtual channel buffers. ACKs and NAKs are the DLLPs that are used for acknowledging successful receptions of TLPs, and for reporting faults in transactions. When a DLLP is sent down to the physical layer, extra sections marking the start and the end of the packet to be transmitted are added. This is illustrated in figure 2.9, note that the outer segments are equal to the outer physical segments of the TLP. The DLLP is not explained in detail here as the RC TLM is to be made in a higher abstraction level.

### PCIe Device Layers

Figure 2.10 shows a PCIe device and its device layers put together in a lego-like structure to form the PCIe Protocol stack. Each layer is described below with extra effort is put into the the higher abstraction layers. The importance of the paragraphs are reflected in their lengths.

### Software Layer

The software layer is where the device’s system logic lies. It is the software application running on hardware on top of the PCIe protocol stack. For an RC TLM this would be the root complex core logic, and for an endpoint this would be the endpoint core logic. The top section of figure 2.10 shows that the all layers are divided into a transmit side and a receive side, even the software layer is

<b>Completion Header field</b>	<b>Explanation</b>
Completer ID	The completer ID tells the receiver of the completion package, who sent it, this has no purpose for routing the packet, however it might come in handy when debugging packet traffic.
Completion Status	The Completion status in the header marks whether the completion package is the result of a successful completion or not.
Byte Count Modified (BCM)	The BCM field is only set to 1 by PCI-x completers, it indicates whether the Byte Count field tells the receiver of the packet the number of bytes in the first transfer payload rather than the remaining payload.
Byte Count	The Byte Count reflects the remaining number of bytes until the request that was the origin of the completion is satisfied, this counter includes the payload of the current completion packet. The field is useful when multiple completion packets are returned to one request.
Requester ID	The Requester ID field in the completion packet represents the device ID of the requester that originated the completion packet, the request field is used for routing the completion packet back to the correct requester by using ID Routing.
Tag	The Tag of the completion packet is equal to the tag of the corresponding request packet, the tag is used by the receiver of the completion packet to remove the request from the outbound request buffer to prevent further re-playing if requests.
Lower Address	The Lower address of the completion header represents the lower 7 bits of the address for the first byte of data returned that was returned to a reader. The lower address is used for determining the next legal Read completion boundary for large read requests as a precaution to cache line alignments.

**Table 2.5:** The completion-header specific fields, explained

slightly divided. Whenever the device core decides to send a packet on the PCIe link, the transmit side of the software layer sends the necessary information to build that packet, down to the transaction layer. The information needed to build a TLP includes TLP-type, address, data, amount of data to be transferred, traffic class and more. The receive side of the device core is responsible for receiving data that has made its way up to the transaction layer, this data includes most data in the normal TLP core except from the ECRC field which is removed by the transaction layer.

### **The Transaction Layer**

The transaction layer is responsible for generating outbound traffic and for decoding inbound TLP traffic. The transaction layer is as shown in figure 2.12 also bipartite like the software layer. One part is transmitting, it generates the outbound TLP headers and assembles the TLP cores on behalf of the software



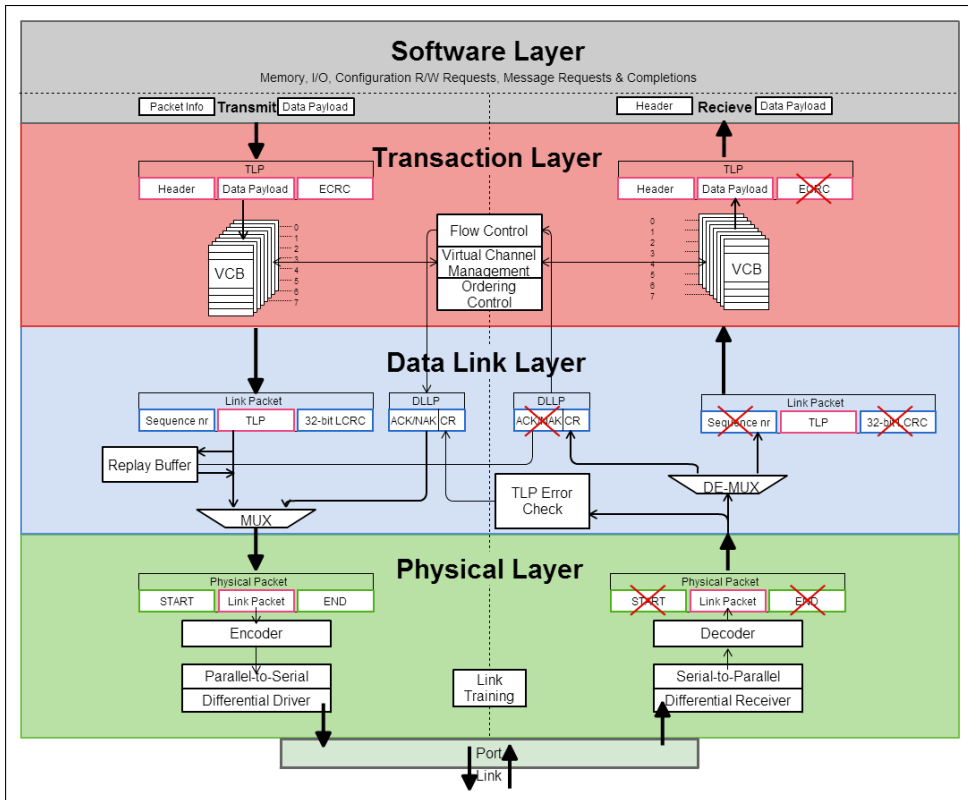


Figure 2.10: The layered structure of the PCIe protocol stack

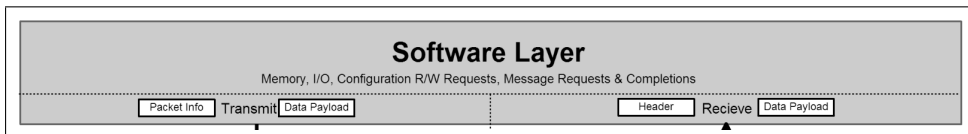


Figure 2.11: The software layer of the PCIe Protocol

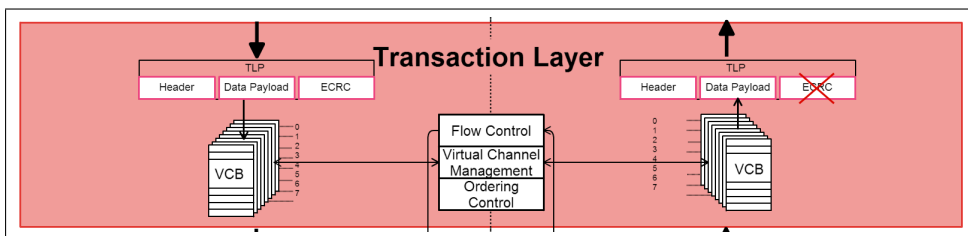


Figure 2.12: The transaction layer of the PCIe Protocol

layer, data payload is encapsulated within the header and an end-to-end cyclic redundancy check field (ECRC). The packets created are then placed in transmit buffers to be forwarded down to the data link layer. The receiving part of the

transaction layer receives packets from the data link layer and places them in receive buffers for handling. When TLPs are picked out for handling they are first checked for CRC errors based on the optional ECRC section of the TLP, if there are none, the packets are stripped for their ECRC field and the resulting information is forwarded to the software-layer together with the data payloads of the transactions.

The transmit and receive buffers in the transaction layer are known as virtual channel buffers (VCBs). There is one transmit and one receive buffer for each traffic class. The 8 VCBs are shown in figure 2.12. All TLPs are marked with a traffic class, a number between 0 and 7, as explained in section 2.1.2. The traffic class is the priority number of the packet, and depending on what TC number the packet has, the packet is put into different buffers with different priorities. The higher the traffic class the higher is the priority of the buffer arbitration. The buffers with higher priorities are cleared at quicker rates than the buffers with low priorities. Quality of Service (QoS) is also ensured by having a credit based flow control to avoid buffer overflow for the VCBs. A virtual channel send buffer is not allowed to be handled unless the credit-status of that specific virtual channel buffer says so. A TLP will only be transmitted if the transmitter knows for sure that the receiver has buffer space to accept the transmitted TLP. The credits represents buffer-space, and they are updated through data link level packets that are sent periodically between devices. The DLLPs are responsible for updating the credit information.

Whenever the transaction layer receives a completion TLP from another device, the tag of the TLP is associated with the tags of previously sent non-posted requests that are stored in an outbound request buffer. The packet is rejected if tag is nowhere to be found in the outbound request buffer. The request stored in the outbound request buffer is removed whenever a completion is received that matches the tag of the stored request. If the request is lost, the packet is replayed after a specified time out period.

Configuration is only covered partly in this assignment as a performance based device simulation targets an already configured PCIe System. A device's configuration registers are associated with the transaction layer of the each device function, a device may have up to 8 functions. Each device function implements a set of configuration registers. The registers are configurable and are used to store system settings such as address maps, device IDs, link capabilities and other system information that needs to be accessed in runtime. Each PCIe device has a set of Configuration registers that are initialized during system boot up by the RC in runtime software. The transaction layer and the software layer access the configuration registers in runtime when sending and receiving packets. The configuration registers are set during system configuration with configuration packets from the PCIe RC.

As figure 2.13 shows, the PCIe device's configuration register contains a 256

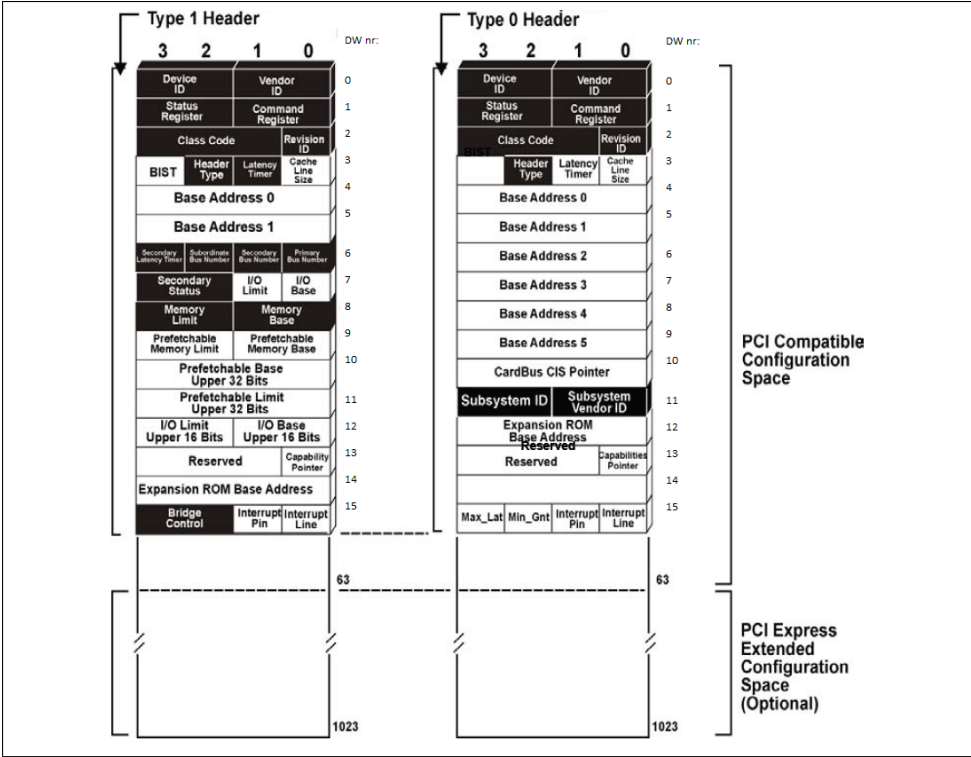


Figure 2.13: Type 1 and type 0 configuration register headers[3]

byte PCI compatible fraction that includes device ID, base addresses and such. The remaining 3840 bytes of the configuration register is dedicated to PCIe configurations, they includes virtual channel capabilities to support QoS and other optional PCIe specific extended capability registers. PCI does not support QoS in the same degree as PCIe does.

There are two different configuration spaces, these are the type 0 and type 1 headers shown in figure 2.13. All devices that require system memory, IO, or memory mapped IO target addresses implements Base Address registers (BARs). Type 0 registers have 6 BARs and are the non-bridge/non-switch format, meaning that endpoints use type 0 configuration registers. Type 1 configuration registers have 2 bars and are used for forwarding packets in switches and bridges. These registers contain unique segments of the system memory address space. The device that owns the BARs, also owns the segment in the system memory and is allowed to respond as a completer whenever requests that are address routed, target an address in this segment. 2 adjacent base address registers may be used to create the support for 64 bit addressing.

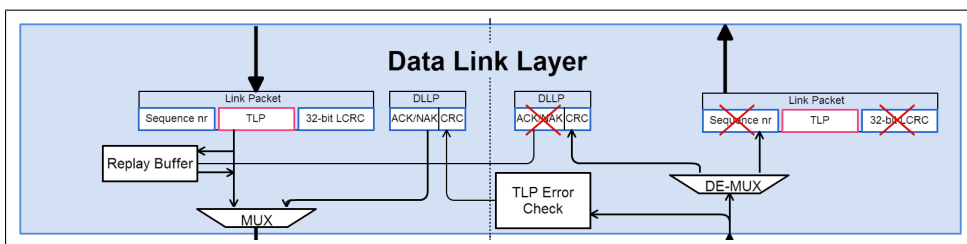
**Transaction Layer rules for building and decoding TLPs.** 6 Important rules for assembling data from the software layer into transaction layer packets and disassembling TLPs that are received from the data link layer are given below:

- Completions may be broken into multiple packets, however the total payload has to be equal to the size of the original request and each completion has to be returned in an increasing address-order. Completions from multiple requests are on the other hand not allowed to be merged such that several requests give one large completion.
- The read completion boundary (RCB) ensures that single memory read request results in several completions. Each intermediate transaction must end with naturally aligned 64 or 128 bytes address boundaries. Only the Root complex is able to have a read completion boundary on 64 bytes or 128 bytes, all other PCIe devices have an RCB equal to 128 bytes. The RCB is a direct result of the cache line size used in memory systems. PCIe performance is thus related to cache-lines. An example containing multiple completions as a result of one memory read request for 192 bytes from address nr. 0x10030 is given here:
  1. 0x10030 -> 0x1003F (0x10 bytes)(16bytes, resulting address pointer is dividable by rcb = 0x40 (64))
  2. 0x10040 -> 0x1007F (0x40 bytes)(64bytes) resulting address pointer is dividable by rcb = 0x40 (64)
  3. 0x10080 -> 0x100BF (0x40 bytes)(64bytes) resulting address pointer is dividable by rcb = 0x40 (64)
  4. 0x100C0 -> 0x100EF (0x30 bytes)(48 bytes)resulting address pointer is not dividable by rcb = 0x40(64)
- Multiple completions as the results of RCB splitting of a single request are allowed to be combined into one completion if the completions that arrived at the egress port queue first have to await sending due to port arbitration delays. If multiple completions are ready for sending, they might as well be combined into one TLP embodiment as long as they satisfy the maximum payload size [8].
- Any completion with a completion status other than “successful completion” will terminate the transaction. A completion without any corresponding request in the outbound request buffer will be handled as an error as they are not expected.
- The maximum payload per TLP is for requests, limited by the value of the Max\_Payload\_size register located in the device’s control register. If it crosses this upper limit, then it is considered a malformed TLP, and is

rejected by the receiver. A large `max_payload` size allows for system communication with less overhead. But it also has its disadvantages of more packet congestion due to full buffers, a detailed description for setting the optimal maximum payload value is given in [25]. The total PCIe system is limited by its weakest components, if a system has a component with a maximum payload of 128 bytes while all other components have 256 bytes, the entire system will be configured by the RC to use 128 bytes.

- Header lengths and ECRC lengths are not included in the length-fields of headers, only data payload is included in this information. Receivers must check that the length field corresponds to the data that is transferred, if it does not match, then the TLP is considered to be malformed. The length field has a double word resolution, byte resolution on reads and writes is achieved with the last and first `dw-byte enable` bits.

### The Data Link Layer



**Figure 2.14:** The data link layer of the PCIe Protocol

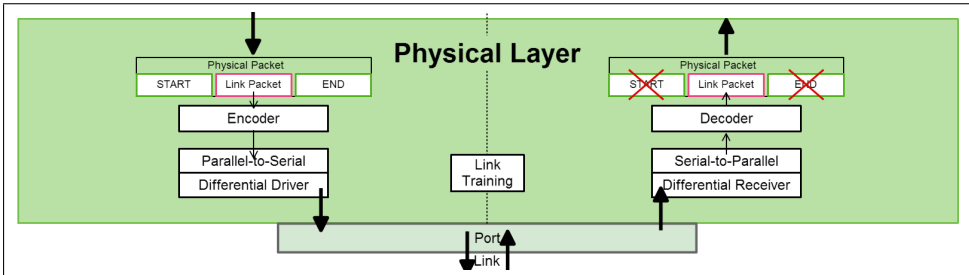
The main function of the data link layer is to ensure data integrity in packets when transmitting and receiving TLPs. The DLLPs are assembled and disassembled in the data link layer. If TLP-CRC errors are detected in a receiver's data link layer, then a NAK DLLP is sent back to the transmitter of the TLP to let it know that an error has occurred and the packet-sending needs to be replayed. The ACK-NAK protocol together with the automatic replay functionality of the data link layer provides a very high probability that the TLP will make it to its final destination unscrambled. Such a feature is ideal for server systems that require low error rate and high availability.

Flow control of the system is ensured using DLLPs. The status of the virtual channel buffers of the transaction layer is updated by periodically sending VC credit information between interconnected devices.

In addition to sending separate data link layer packets, the data link layer also concatenates a sequence ID and a LCRC field to each TLP received from the transaction level for further forwarding. The sequence id is used to identify

packets in the replay buffer with incoming ACK/NAK DLLPs, and the LCRC field is used for deciding whether to return an ACK or a NAK DLLP to the sender.

## The Physical Layer



**Figure 2.15:** The physical layer of the PCIe Protocol

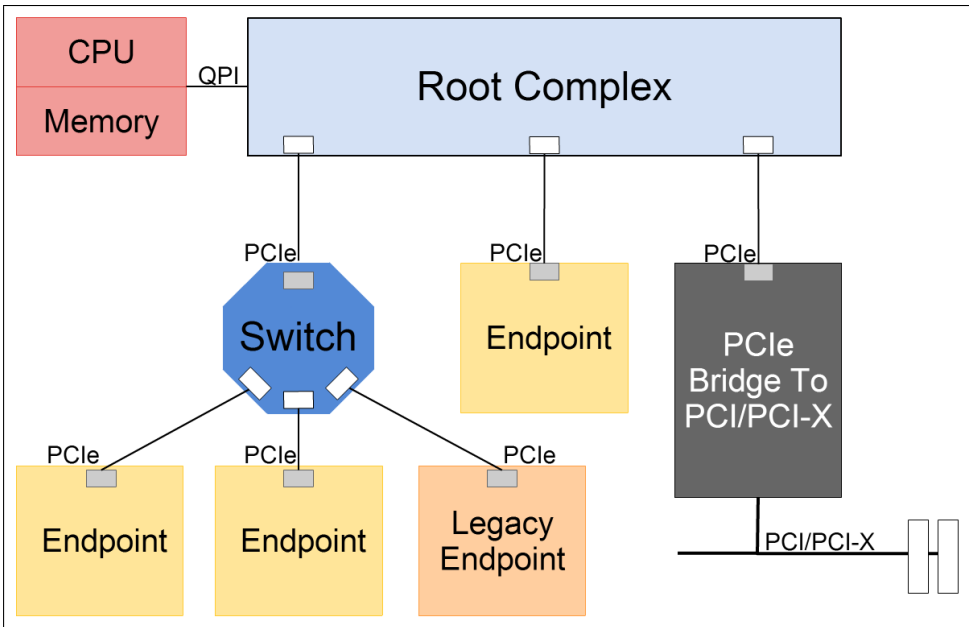
The layer that has direct contact with other PCIe devices is called the physical layer. This is the final layer that the packet has to pass through for it to leave the PCIe device, or the first layer that an incoming packet will face. The physical layer forwards both DLLPs and TLPs, to which it adds start of packet and end of packet indicators. There are two levels in the physical layer, one is the logical part and one is the electrical part of the layer. The first one processes the packets before sending them, to or from the physical electrical part, depending on the traffic direction. The physical electrical part is the analog interface for the physical layer to the link.

The packets leaving the Physical layer towards the egress port are converted to a serial bit-stream before they are sent out on the wire lanes. All PCIe packets are encoded using a 8/10 bit encoding, meaning that 2 extra bits are added per 8 bits of TLP or DLLP data. These 2 redundant bits are used for embedding a clock into the serial bit stream, removing the need for a clock tree.

A PCIe link between two devices that deploys the PCIe architecture is either of type 1x, 2x, 4x, 8x, 16x or 32x. The type number denote the number of two-directional lanes that exists in the connection link between the two devices. The more lanes implemented on a link, the faster a packet is transmitted, thus the greater the bandwidth of the link. A lane consists of four signal-wire pairs, two wires for each transfer-direction. Each of these signal-wire pairs use differential signaling to achieve transfer rates of up to 2.5 GT/s for PCIe gen1, 5 GT/s for PCIe gen2, 8 GT/s for PCIe gen 3, and an announced transfer rate of 16 GT/s for the future PCIe gen 4. One wire in the pair represents a positive differential terminal, and the other wire represents the negative terminal. Whenever the voltage difference between the two terminals is positive a logical 1 is interpreted by the receiver, and whenever the difference is negative a logical 0 is received. The dif-

ferential pair is in its high-impedance tristate condition or idle-state whenever the voltage difference is 0. A signal event is triggered whenever the peak to peak voltage difference is between 800mV and 1200 mV.

### 2.1.3 The PCIe Topology



**Figure 2.16:** Topology of a typical PCIe system

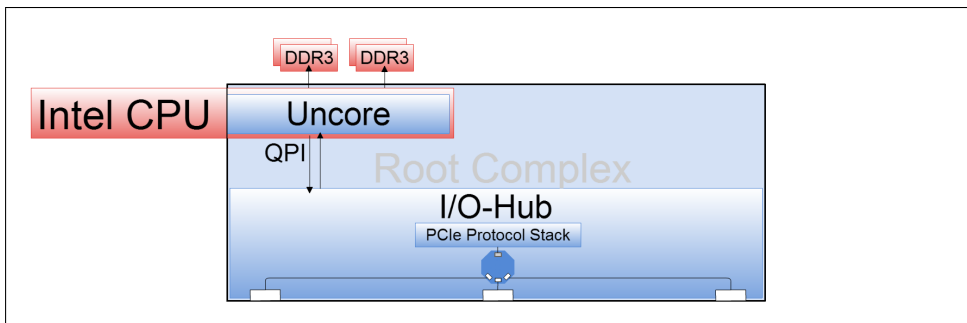
Figure 2.16 shows the component structure of a typical PCIe system. Endpoint devices and switches are connected to the memory and the CPU via the module known as the Root Complex. Whenever the CPU desires to communicate with peripherals, it performs transactions via the RC. Same goes for the endpoints, whenever they want to access data that is located in the subsystem memory next to the CPU, they have to send request TLPs to the root node in the PCIe tree for handling.

#### The Root Complex (RC)

The RC interconnects the CPU and the memory to the PCIe switch fabric. It is similar to the host bridge for PCI. The RC is a critical component in the PCIe topology as it serves as the root node for the hierarchical PCIe tree, connecting the tree to the host CPU and Memory. An RC includes multiple components such as a memory interface and a processor interface for single CPUs or multiple CPUs. The CPU and the memory are connected to the RC using local bus architectures in a manner that allows the root complex to send transaction requests to PCIe devices on behalf of the CPU. The RC is also able to perform DMA, freeing the CPU as an intermediary, accessing the host memory directly.

The buses that connects the PCIe tree to the CPU are company specific because the RC is built into the chip-set for the CPU. Intel CPUs have been using





**Figure 2.17:** The PCIe multiport RC

the bus architecture known as the front side bus (FSB) for a couple of decades. The FSB in Xeon and Itanium was replaced by the QuickPath Interconnect (QPI) [16] [28] for newer Intel micro-architectures such as Nehalem, Westmere and Xeon arriving after Xeon Bloomfield, starting in 2008-2009. The QPI is displayed in figure 2.17. It is a point to point packet based interconnect located within the Root complex, interconnecting the I/O Hub and the Uncore [12] [31] of recent Intel architectures.

The root complex is in addition to serving as a bridge between peripherals and core logic, also responsible for controlling hot plug of new PCIe devices, configuration, power management, interrupts involving the CPU, error detection and for reporting system messages. As explained earlier, each PCIe device contains configuration registers, these configuration registers are set by the root complex during system initialization and hot plug using configuration packets. The configuration registers contain data required to route packets to their appropriate destination. All PCIe devices in a system are initialized with device IDs consisting of bus numbers, device number and function numbers. The device IDs are used for ID routing by marking the packets with the corresponding sender or receiver.

There is only one RC in a PCIe network, and it is always initialized with the device ID; bus nr: 0, device nr: 0 and function nr 0. It can have one or several PCIe devices connected to it, either directly or indirectly via switches. Several connections directly to the RC requires that a switch is implemented as an internal submodule. All PCIe packet-traffic heading towards the RC is considered to move upstream, and all traffic moving away from the RC is considered to move downstream.

Multi-port PCIe RCs have the same requirements as a switch with respect to arbitration and timing requirements. Arbitration algorithms, and timing requirements are not defined in detail in PCIe RC specs, so these are vendor specific. However, a PCIe RC realization must satisfy all the paragraphs in the checklist given in [35] in order to be a valid PCIe RC. For this thesis however, only a

delay-based model is to be provided, so the checklist is only relevant as theoretical background information.

**Snooping** A phenomenon called snooping plays an important role on RC performance, and thus the PCIe system performance. All memory accesses to a PCIe system memory complex are considered cache-able by the host at any time. This means that the CPU continuously stores copies of system memory in its caches. An EP requesting memory access to a system memory segment will experience a wait period for the RC to search in processor caches for copies of the memory segment. The CPU might have modified local copies with the intention to update the memory location. If this is the case, then the requesting EP will have to await the memory update before it is allowed to access the memory. The RC's cache check, filters out violations of cache coherency. Cache coherency is crucial, yet time consuming. The time it takes to snoop the CPU cache is bounded below a maximum limit, but it is still not predictable, largely influencing the Jitter of the MRd-CplD completion delay. [22]

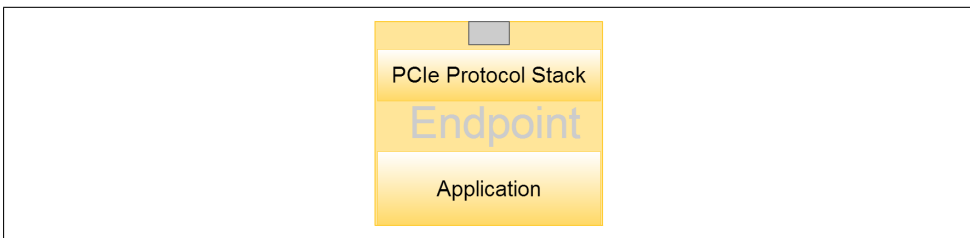
A way to prevent system jitter from snooping is to dedicate uncacheable areas of system memory. In this way the CPU will avoid caching those areas. A second way is to have the software layer to set the "No Snoop" attribute in the TLP as described in 2.1.2. This causes the host to skip snooping regardless of its previous instructions, it can only be done when the software guarantees that the memory segment is nowhere to be found in the CPU cache. The requester knows that this will not cause any conflicts. For high priority virtual channels that requires isochronous paths, jitter can be avoided by setting configuring the RC to reject all incoming Snooping transactions on that channel. All packets that does not have the "No Snoop" attribute set might be rejected.

**Relaxed Ordering** A phenomenon called Ordering also plays an important role on RC performance. It is often beneficiary to have a strict ordering of packets that have data dependencies or memory location dependencies. A typical example would be a movie stream, you want packets to arrive in the order of the movie. However, a strict ordering of packets is not always needed. Successive transactions that do not have dependencies between each-other might allow a relaxed ordering of the transfers. Forcing ordering of packets that are independent can cause large loss of bandwidth, packets may clump up waiting for their predecessor to be sent. Instead the packets are allowed to jump the queues, and to be sent once they are ready. Relaxed ordering is set by setting the attribute in the TLP Header called "Relaxed Ordering" as described in 2.1.2.

**TLP Hints, Direct Cache Transfer** A new feature for PCIe Generation 3 is the phenomenon of TLP Hints. It turns out play an important role on RC perfor-

mance, using caching to handle inbound memory transactions. One of the redundant bits in the generic TLP header fields in 2.1.2 is swapped out with a TH bit. If set correctly, the bit provides the system with an idea of how to handle the TLP in an optimal fashion to reduce latency and congestion [22]. When set, it means that a packet from an EP to the host memory is to be added to a cache in the RC with the intention of being read by the CPU as fast as possible. Instead of wasting time on writing to the memory and then having the CPU reading it from the memory again, the RC writes to the Cache. Software or interruptions notify the CPU that it has a "mail". TLP hints leads to faster access time and reduced traffic targeting the system memory.

### Endpoint (EP)

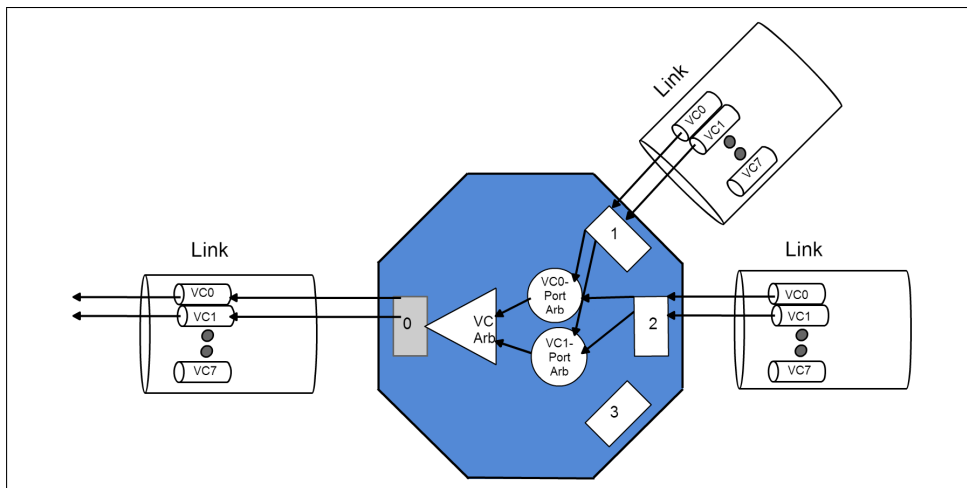


**Figure 2.18:** The PCIe Endpoint

A PCIe interconnected network can have several concurrently connected endpoints as illustrated in figure 2.16. A PCIe endpoint can either be a requester or a completer. It can initiate a transaction as a requester and it can also answer a request with a completion transaction. There are two types of endpoints, legacy endpoints and normal endpoints. Legacy endpoints are endpoints that were designed for an other architecture, typically PCI-X and was redesigned to fit that of PCIe, keeping the device core. Legacy endpoints are allowed to use I/O transactions, and are free to choose at design time whether they want to support a 64-bit address space or not. Normal PCIe endpoints do not have to support IO or locked transactions. They do however have to support 64 bit addressing. Endpoints are allways configured with the device number 0 on a bus.

### Switch

A PCIe switch is similar to most network switches in LANs, the switches interconnect n packet-based PCIe devices and forwards packets in between them. The PCIe switch has as shown in figure 3.7 an internal bus with its own bus number used for forwarding packets to the appropriate PCItoPCI ports. The PCIe switch reacts differently to the various routing methods earmarked in the header of the incoming packet. There are three different routing methods, these



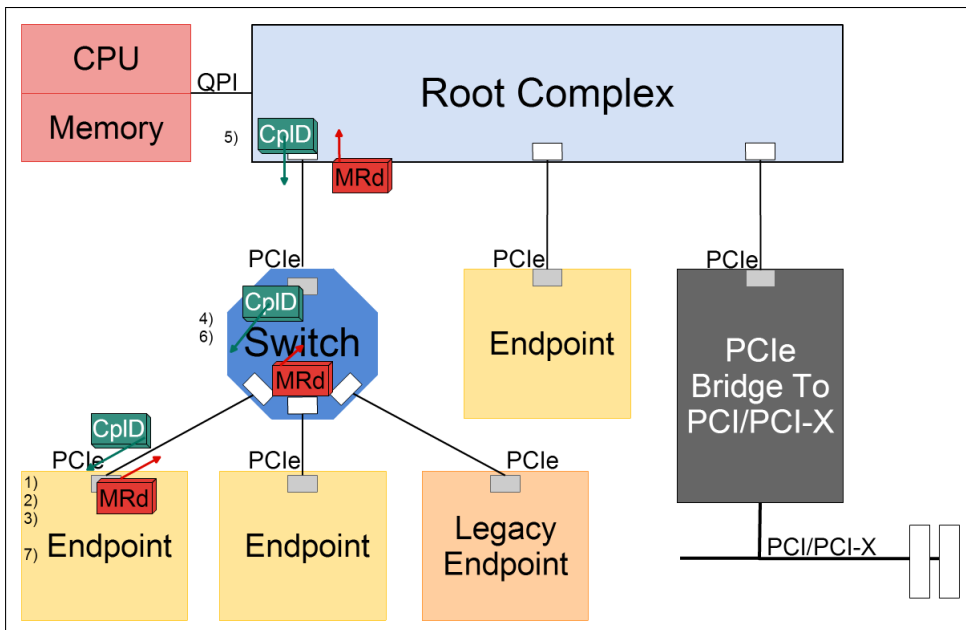
**Figure 2.19:** A packet switch with VCB and Port arbitration

are: Address routing, ID routing and implicit routing, switches have to support forwarding of these three. For Address routing: The switch firstly checks if the switch itself is the rightful receiver of the packet, if not, then the packet is forwarded appropriate port based on the port data located in BARs in the switch's type 1 register header. The switch has one configuration space header for each port. In order to create a multi-port RC a switch might be implemented into the RC. Multi-port switches also have the possibility of priority differentiation using VC buffers, this is made possible by having a common VCB system for each egress port. Packets are forwarded to the rightful ports, and then placed in a VC arbitration buffer for that port.

#### 2.1.4 PCIe Packet Sending Example

Now that the basic building-blocks of the PCIe architecture are explained, a simple complete example of a packet transfer can be given. This example shown in figure 2.20, demonstrates the EP as a requester and the RC as a completer. The EP issues a MRd to system memory and receives a CplD. Details of the example is further explained in the bullets below, the numbers correspond to the numbers in the figure:

1. The software device core of the PCIe device, issues a request for service to the PCIe transaction layer. The exact data that is issued to the transaction layer is device-specific, however it would typically include: command type, start address, transaction type, size of the data payload if any, traffic class for the packet to be sent, and attributes; NoSnoop and Relaxed ordering. The transaction type in this case is MRd.



**Figure 2.20:** Example of a memory read transaction on a PCIe topology

2. The transaction layer constructs the TLP header and adds it to the VCB that corresponds to the TC in the packet header. A copy of the packet is also stored in the Transaction layer's outbound request buffer. The packet is sent down to the data link layer once the VCB arbitrator empties the buffer.
3. Sequence number and link layer CRC is added to the TLP in the data link layer and a copy is stored in the replay buffer. The packet is sent directly down to the physical layer where a start and an end sequence is added to the packet. The packet is then serialized and sent on the link on a specified number of lanes.
4. The switch receives the wrapped up TLP, the tlp is stored in port buffers awaiting port buffer arbitration. Once chosen by the port buffer arbiter, and the incoming VCB arbiter. The switch performs various low level error checks and checks the address field of the packet versus its own BARs in its type 1 configuration register. It finds out that it is not the rightful receiver and forwards the packet the TLP is then added to the egress VCB on the port that fits the status of the configuration register. The TLP is sent further upstream towards the RC once the egress VCB arbiter handles it.
5. The receive part of the RC works in the opposite direction of the send part of the EP. Once the request reaches the device core, memory is read and

the data is returned back to the transaction layer again for building of a completion packet. The request ID in the memory read request is added to the completion packet, so is the outbound request tag and the data payload is encapsulated.

6. The packet is sent to the switch which uses ID routing to forward the completion packet back to the requesting endpoint.
7. Upon reception the inverse approach as that of sending the request is performed, and once in the transaction layer, after VCB receive arbitration, the packet tag is matched to its outbound request and the outbound request is removed. The data payload is sent to the endpoint's software layer.

## 2.2 PCIe Related Performance Metrics

PCIe system performance is a hot topic of discussion in the following chapters since the goal of this thesis is to create a model to be used performance simulation of a PCIe EP. Parameters that describe PCIe performance are: Bandwidth, latency, cost/power, Quality of service(QoS) and Error protection[25]. This section is dedicated to clarification of the performance parameters in PCIe that are most relevant for this thesis, PCIe bandwidth and latency.

### 2.2.1 Bit Rate / Bandwidth

Bit rate or bandwidth is the number of bits or data processed per unit of time. There might be several interpretations of what to include in the bit rate of a system, some of these are explained below.

#### **Gross bit rate/ raw bit rate/ gross data transfer rate/uncoded transmission rate**

Are all names for the physical data bit rate that is transferred per second over a communication connection. This performance metric includes redundant data as well as the data of interest. Using PCIe as an example, baud rate includes the bits used for error checks in the TLP. [bit/second]

#### **Symbol rate/baud rate/modulation rate**

Baud rate, symbol rate or modulation rate is the number of symbols transferred per second. Depending on what the symbol size is, the baud(Bd) rate might vary from the bit rate. The baud rate is only equal to the bit rate when the symbol size is equal to one bit. The symbol size might also be less than one bit, this often

occurs in low speed encrypted channels, or in communication channels with a high amount of error-check redundancy, (like PCI-e). [symbols/second]

**Net bitrate / information rate/ useful bit rate/payload rate / net data transfer rate / coded transmission rate / effective data rate / wire speed**

Net bit-rate has many synonyms, they all refer to the capacity of a communication channel excluding the physical layer protocol overhead, redundant error-check-codes, and general channel encoding. Net bit-rate always satisfy the inequality net bit-rate  $\leq$  gross bit-rate. [bits/second]

**Peak Bit-rate /Max Bandwidth**

It is the peak potential number of bits per seconds that a communication channel can obtain. There is high risk of confusion when dealing with peak bit-rate, whether the communication-potential is listed in gross, or net bit-rate. This paper talks of peak bit-rate as maximum net bit-rate communicated unless otherwise is stated. [bits/second]

**2.2.2 Latency and Jitter/Packet Delay Variation(PDV)**

**Latency**

Latency is the delay it takes from the start of an action to the completion of the same action. For a PCIe switch, the latency is the time it takes from the start of packet (SoP) on an input-pin to the SoP on the output-pin on another switch-port [17]. When talking about latency involving several communicating devices, one can differ between one-way delay and round-trip delay. One way delay is the latency it takes for a communication unit to travel from one device to another. Round-trip delay is the time it takes from when something was spoken by one device until the response from the device that was spoken to is returned.

When observing latency from a requesting PCIe-EP's point of view, the latency would typically include the round-trip delay of forwarding the request to the RC, handling the request and also routing it back again to the endpoint. The latency is composed of wire propagation-delays, switching-delays, priority-based delays, routing-delays, request handling delays and if the request is non-posted, the latency will also include the corresponding delays for routing the completion packet back to the requester. This occurrence is illustrated in figure 2.21, all delays can be modeled by a total MRd-CplD delay. The intention of this figure is to show that there are several layers of delay-details that can be merged and included in a more coarse grained model Z(mrd-cpld total) represented by only one value. This course grained delay value is adequate for modeling system delays.

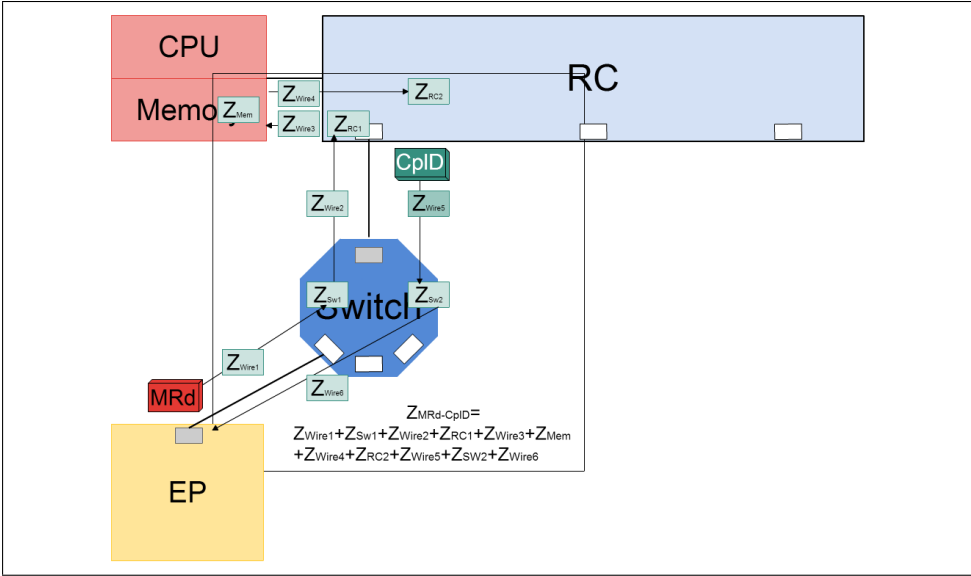


Figure 2.21: MRd-CplD delays, decomposed

**Jitter/Packet Delay Variation**

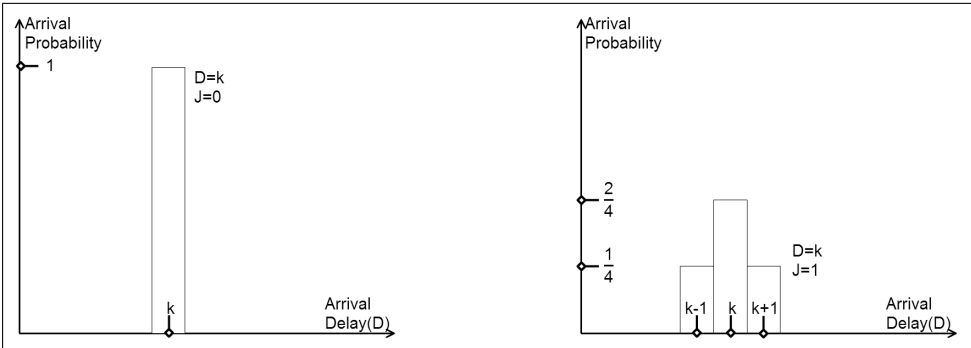


Figure 2.22: Jitter demonstrated, jitter equal to zero vs jitter equal to 1

When talking about jitter in computer networks, one refers to the deviation from exact periodicity in latency. A simple example of jitter is given in figure 2.22 A theoretical network where the latency is constant implies that the network has no jitter. This means the uncertainty of latency, or how stretched the distribution of arrival latency is, seen from a statistical point of view. Stretching of distributions is called statistical dispersion. Jitter is a term with multiple meanings, a better term to use for oscillations in packet arrival delay is Packet Delay Variation(PDV)[9]. PDV is defined as the difference in end to end delay excluding lost packets. Instantaneous packet delay variation (IPDV) is the delay variation between two successive packets. If PCIe packets are transmitted every

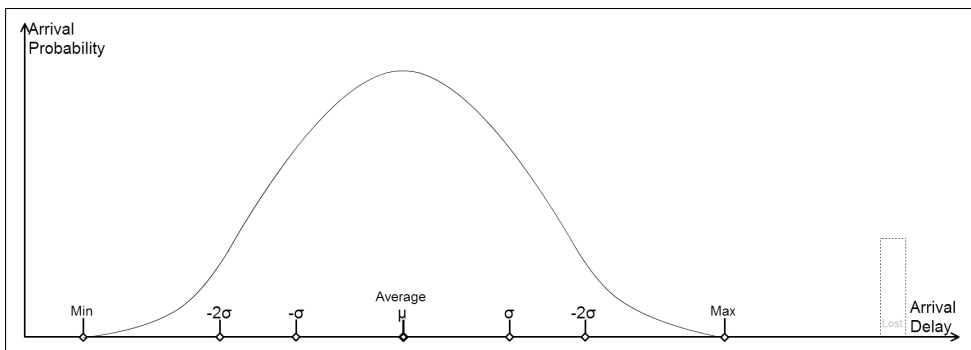


200ns in a system and a packet is received 300 ns after the previous packet was received, IPDV=-100ns, and dispersion is said to have occurred. As a contrast, whenever IPDV is positive, clumping is said to have occurred. An example of clumping would be a packet arriving 100 ns after its prior in the previous PCIe Example. Clumping might lead to buffer overflow and packet loss which has to be avoided in order to ensure QoS in the system. One way of solving clumping is to increase receiver buffer-space in a system.

Typical causes of jitter are:

- Large networks, many plausible causes of jitter. In a PCIe system this might be many switches between an endpoint and the RC.
- Priority stalls
- Variable processing delays
- Buffer bloat, some equipment have been designed with over-sized buffers that can be filled up when packet congestion occurs.

### Representing Latency and PDV



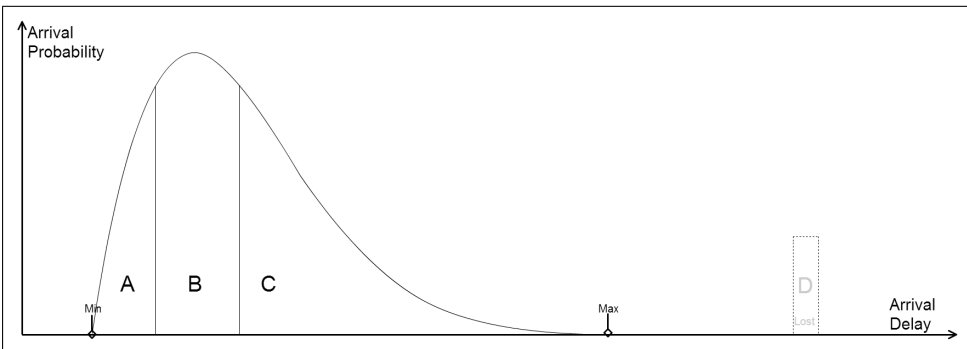
**Figure 2.23:** Packet arrival time, normally distributed

Latency and jitter can be represented by Latency distributions. A common way to represent the jitter variable in packet based networks such as PCIe is by the use of the normal distribution. The average latency is  $\mu$  while the jitter is represented by a number of standard deviations,  $\sigma$ , as the variance of the normal distribution. An example of a jitter-latency distribution is illustrated in figure 2.23. The communication latency is represented by  $x$  variations and the probability for the given communication latency on the y axis. No packets are returned immediately, this is due to the absolute minimum wire delay it takes to transfer data physical communication systems. In packet based systems more

realistic PDV-latency distributions are often more long-tailed, fitting the Poisson distribution. Long tailed distribution are also known as heavy tailed distributions. There are only empirical evidence for the reasons behind Long tailed distributions in packet traffic. There are three main observed reasons for these skews [29] in packet based communication channels, these are:

- Variations in application layer in the form of distributions of file sizes to be transferred and so on.
- The transaction layer in the PCIe layer implements flow control. Congestion avoidance leads to delayed transfers for a small fraction of the packet majority.
- Also in the transaction layer of the PCIe, a critical packet creation rate might lead to pink noise on the channel and congestion.

The request-completion latency for packet based protocols that all requesters have to face in order to receive completion packets, is highly dependent on the channel-traffic, the load situation of the system, as well as the request itself. Requests might be semi-starved in priority buffers, they might also be forced to wait for memory availability and so on. This variable nature gives a more long-tailed latency distribution. The long tailed request-completion delay is shown in the sketched graph in figure 2.24.



**Figure 2.24:** Packet arrival time, heavy-tailed distributed

A few request packets are quickly completed because of low system load or high packet priority or both. These requests are referred to in this thesis as “early completers”. Most requests are completed within latency segment B, the requests in this segment will experience some wait-time in buffers due to either medium packet-priority or varying queue depths in VCBs, these requests are referred to in this thesis as “average completers”. Some non-posted requests are completed late, segment C in the figure represents the “late-completers”. The late completers are often the result of low priority queue starvation which

can be caused by heavy traffic in the higher priority queues. For PCIe, the arbitration algorithm depends on the actual implementation in the system, so this can be avoided, or at-least drastically reduced by implementing larger buffers, or smarter arbitration algorithms. The last segment are packets that are never completed. Segment D represents the packet loss due to packet congestion and thus over-flooded, packet loss might also be caused by bit flips in the physical layer. These are referred to in this thesis as “Never completers” or simply “lost packets”. The never completers are like any other PCIe request stored in the outbound request buffer. This means that after a specified outbound request delay, the request is refreshed. And if the circumstances reasonable, the request will be answered within the normal completer-delay.

Multiple probability distributions can be super-positioned into one by using the principle of mixture distributions[21]. This can be done by weighting each sub-distribution according to their respective portion of the total sample number and then adding all distributions together. This sums up to a weight equal to one as shown in the following formula.  $P_i$  is the  $i$ 'th of  $n$  probability distributions and  $w_i$  is its corresponding weight on the total system distribution.

$$f_{total}(\dots) = \sum_{i=0}^n P_i(\dots) w_i$$

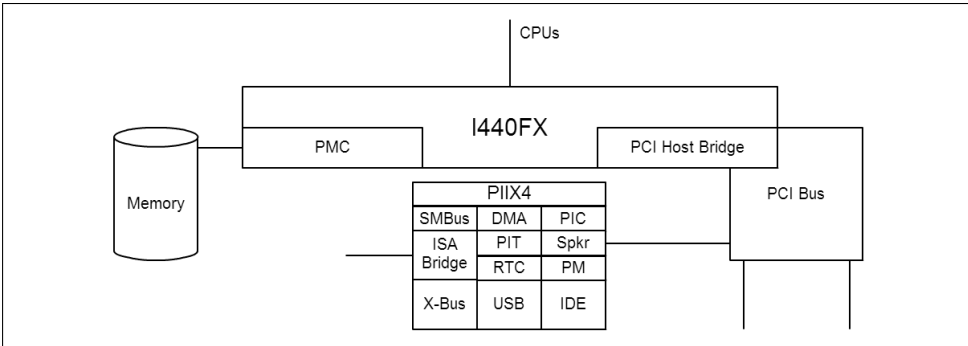
**Figure 2.25:** Weighted mixture of probability distributions

## 2.3 Root Complex Emulation with QEMU

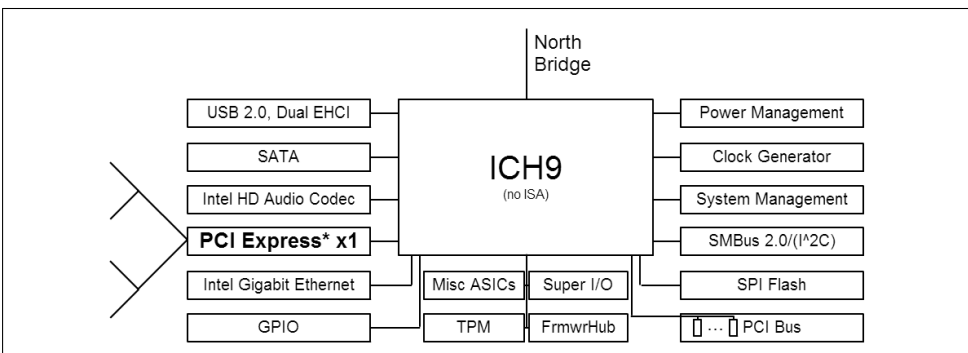
QEMU is short for Quick Emulator and is a free, open-source software made in Japan. It is a hosted hypervisor that creates a virtual computer hardware platform. QEMU is short for “Quick EMUlator”, which describes what it does. It emulates CPUs, translating one instruction set into another via binary machine code. This method is known as dynamic binary translation. In addition to emulating CPUs QEMU also provides a set of device models that enables it to run operating systems on top of the emulated hardware without modifying the OSs.

This platform has the ability to emulate hardware and to run different hardware-specific operating systems on top of this emulated hardware. QEMU proves to be a useful tool when testing software applications, testing hardware performance, trying out operating systems and when running tasks that are unable to execute on a target platform.

Recently a new chipset was introduced as a launch option for the QEMU emulator. I440FX, the old chipset, is out of date and does not use PCI-express as well as other important features, instead it only supports the PCI bus. Q35 is the model-name of the new chipset from intel that is implemented in QEMU to support PCI-E pass-through, that is, to allow guests to have exclusive access



**Figure 2.26:** Intel's I440fx supports PCI and utilize ISA, available in QEMU



**Figure 2.27:** Intel's ich9 supports PCIe, now available in QEMU

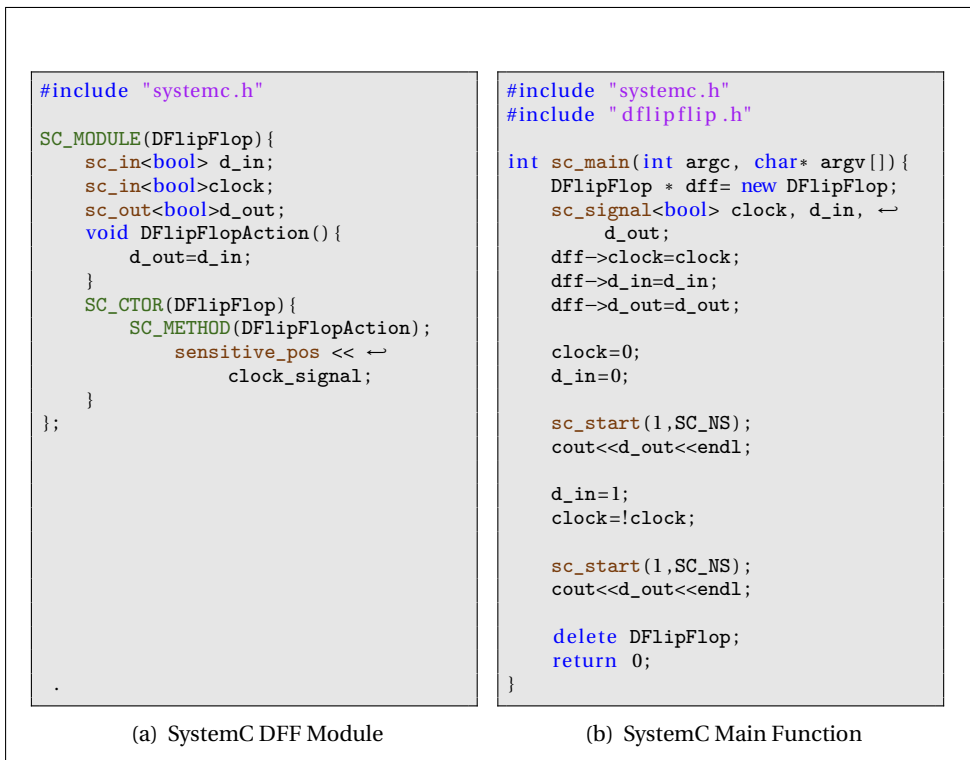
to PCI-e devices for a range of tasks. PCI-E devices are allowed to appear and behave as if they were physically connected to the emulator. The Q35 chipset was released in 2007 and is thus not exactly new, however it was chosen because it is well established, it has PCIe support, as well as usb, sound, ahci and bridges. Once Q35 is added as a HW option, adding new chipsets will be easier as the code surrounding the old I440FX is rewritten in a more flexible manner. The Q35 consists of a north bridge, MCH and a south bridge ICH9, the ICH9 contains an integrated advanced host controller. The pictures in figure 2.26 and 2.27 show that by changing from i440fx to the Q35 PCIe is unlocked in the simulation environment.

The QEMU emulator is developed using git and is available for download on QEMU's webpage [36]. The emulator welcomes contributions either for fixing existing bugs or for adding new functionality. The software contributions have to be created following the set of rules given in [37], it should then be issued to a QEMU mailing list for approval. The open-source feature of the emulator opens up the possibility of customization for simulation purposes.

## 2.4 SystemC

This chapter is not meant to be a complete description of the SystemC library, just a basic introduction for understanding the implementation approach of the root complex model.

SystemC is a coding framework that provides a discrete-time and event-driven simulation environment. SystemC is implemented as a library extension of C++, with macros and classes describing the discrete notion of time, module communication and a concurrent process functionality. Its wide range of abstraction-level-possibilities allows design-engineers to use it for hardware and software co-design. The fact that SystemC is a C++ library, makes it a superset of C++, all C++ code works with SystemC code, and you are actually writing C++ code when you are describing the module processes in SystemC. The SystemC library includes a simulation kernel that allows evaluation of system behavior. SystemC is ideal for transaction level modeling of systems(TLM) because of its great range of expression through object-oriented design partitioning and templates. VHDL and Verilog are better as detailed hardware descriptive languages(HDL) due to SystemC's syntactical C++ overhead.



**Figure 2.28:** A SystemC D-FlipFlop example

The basic building blocks of SystemC are modules. An example of a SystemC module is the D flip-flop module given in the code snippet in figure 2.28(a). SystemC modules have IO-ports described with internal methods and attributes `sc_port` derivatives such as: `sc_in` and `sc_out` templates. Modules are macros containing the class-structure of C++, they are allowed to have member attributes and methods just like the average c++ class. IO ports and signals in SystemC are implemented using templates to provide flexibility on the data type that is carried by the port or signal. Even user defined data types are allowed on the IO ports. This eases the communication between modules in a well-presented manner. Whenever ports are created to carry user-defined structures, the user-defined structure needs to implement certain functionalities that is required for sending, receiving and monitoring the port-signal. The functions that has to be implemented for a user-defined signal-unit are `sc_trace` function for writing trace-waveforms, the assignment operator, the comparison operator and finally the ostream operator.

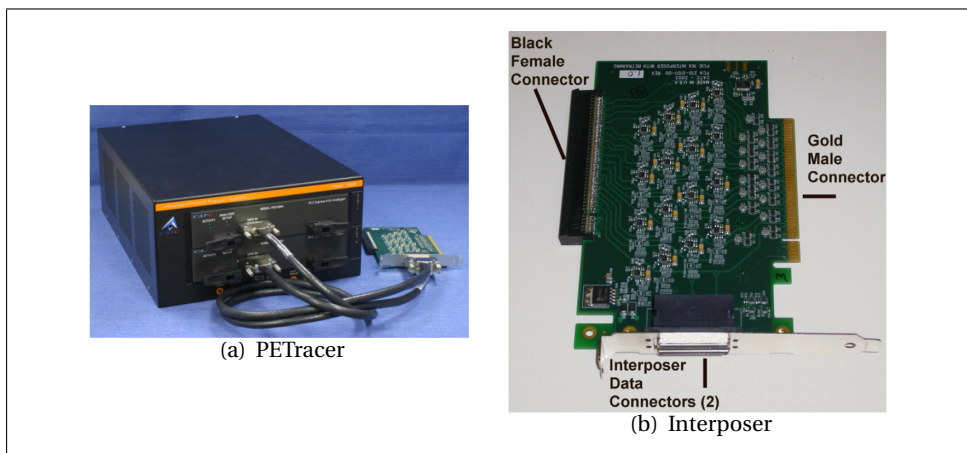
A module's member functions may be declared as internal concurrent processes through the constructor-call of either `SC_METHOD`, `SC_THREAD` or `SC_CTHREAD`. The `SC_METHOD` in 2.28(a) works as a listener, executing the function specified within it once it is triggered by the clock in its sensitive-list. A SystemC process can be made sensitive to signals, events or local variables. `SC_THREAD` processes runs continuously with while loops and wait that holds the thread until certain events occur. The `SC_CTHREAD` processes are described in the constructor of the module with a clock in addition to the function, the clock triggers the wait statement within the process.

SystemC's main function, `sc_main` wraps around the main function in C++ and is the entry point of the program. The difference between `sc_main` and `main.cpp` is that a call to the `sc_main` function also ensures initialization of the simulation kernel, and structures that enables the simulation-nature of SystemC. The command-line arguments `argc` and `argv` may be used in a similar fashion as those of the `main.cpp` function. Before starting the simulation-timer, an instance of the module to simulate has to be instantiated, and its signals has to be tied to outer layer `sc_signals`. The discrete notion of time in SystemC is based on 64 bits unsigned integers. Time is implemented in the data type `sc_time(double, sc_time_unit)`. Each object of `sc_time` contains information of the time resolution in the form of `sc_time_unit` as well as a amount of time quanta. A variable amount of simulation time can be elapsed once `sc_start` is called with the specified simulation-step and the time quantum.

The model created in this thesis, has been created using SystemC version 2.3.0, and the c++ version from 2011. Any C++ compiler can be used for compiling SystemC code, however the SystemC library has to be downloaded, compiled and linked to correctly. To have SystemC harmonizing with Valgrind to prevent false positived of memory loss, the SystemC library needs to be com-

piled with the `-pthread` flag before usage. For the model represented in this thesis gcc version 4.8.1 was used for compiling both the SystemC library and the RC model. More information about the SystemC library can be found in [14] [1] [27].

## 2.5 The LeCroy PCIe Gen. 1 Trace System



**Figure 2.29:** Trace hardware from LeCroy and CATC[20]

To gather trace data from a PCIe system, a trace solution from Teledyne Lecroy can be applied. The trace solution described here allows tracing of a PCIe system with a PCIe card, acting as an endpoint and a motherboard containing the RC and the PCIe interconnect. Teledyne Lecroy is a company that delivers oscilloscopes, protocol analyzers and other measurement equipment that assists a wide range of industries in designing and testing electronic devices of all types. Teledyne Lecroy bought the company CATC in 2004 to obtain CATC's PCIe solutions and other electronics-analyzer tools.

Lecroy's solution for PCIe packet tracing consists of an interposer that is connected to a PCIe Multi-Lane Protocol Analyzer, these are shown in the figures 2.29. The analyzer is called PETracer EML, two of these are docked inside of Lecroy's Universal Protocol Analyzer System (UPAS), and they both utilize the CATC Trace software that allows users to control the process from a PC-workstation.

Using CATC PETrace, real-time triggering and filtering options can be set through a GUI in the windows operating system. The Analyzer feeds live data from the PETracer EML to the PC for real time statistics and recordings. Recorded trace data can be processed and displayed in a systematic manner where one can perform hiding of packet-header fields and collapsing of data payloads and so forth. A snippet of the work-station application is shown in figure 2.30 The packet-log in recorded trace-files that include system timestamps for all packets recorded. The timestamps can be used to find delays between requests and completions. The maximum trace-recording size is limited to 4 Gb in each direction due to the buffer size of the PETracer EML. PCIe generations 1.1, 1.0a and 1.0 are supported using their full speed at 2.5 GHz transfer bit-rate per lane.



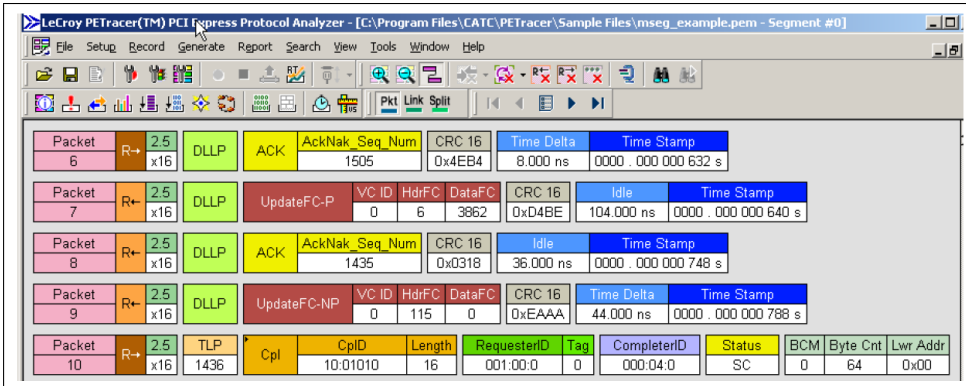


Figure 2.30: PETracer software snippet[20]

Equipment that is used for tracing PCIe traffic with this approach is:

- Universal Protocol Analyzer System.
- 2x CATC, PCIe Analyzers PE-EMLTracer (one for each data direction).
- InterPoser.
- Interposer cables
- Workstation for controlling and recording traces
- PCIe endpoint and a PCIe motherboard to dock the interposer in, typically a server.

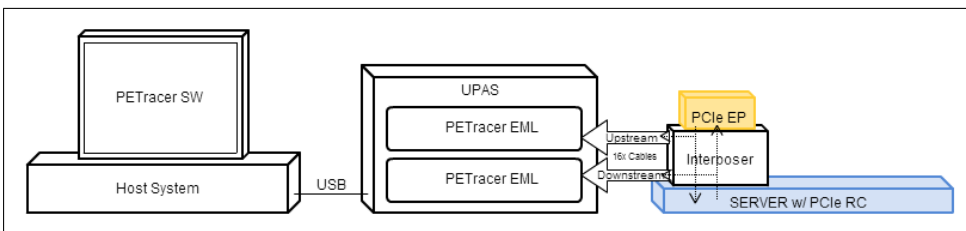


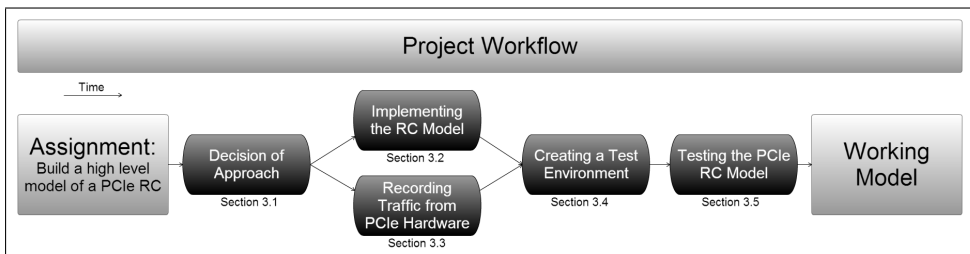
Figure 2.31: A hardware setup for the protocol analyzer

Figure 2.31 shows the hardware-setup of a PCIe trace-scenario. The host system runs a PETracer software that is used both for controlling the data tracing and for analyzing packet data. A USB cable serves as the interface between the analyzing hardware and the analyzing software on host system. The USB cable connects directly to the UPAS system that contains two EML modules for

analyzing bidirectional traffic. One EML module analyzes data that moves upstream towards the RC and one EML module analyzes data that flows downstream, this information is copied and transferred via 16x cables from the interposer to each EML module. The interposer is a male to female PCIe card that is serially connected in between the PCIe server and the PCIe endpoint, giving the EP the illusion that it is directly connected to the RC while eavesdropping on the traffic.

For further details about how the PETracer EML Analyzers work together with the UPAS 100k module, please refer to the PETracer manual [20].

## Methodology



**Figure 3.1:** Outline of the project workflow

Figure 3.1 maps this chapter’s sections to the project workflow. The 5 milestones that are executed towards a working implementation of a high-level model of a PCIe RC are shown in the illustration. Step 1 describes the decision of a systematical approach for modeling an RC. The level of abstraction provided a natural pointer towards selecting the implementation language to be SystemC.

The first milestone unlocked the implementation of the RC TLM itself and also tracing of PCIe traffic. Implementation of the RC could not be completed without the insertion of delay-data from the hardware tracer. Linking of recorded traffic-data to the RC model resulted in a TLM of the PCIe RC that is realistic with respect to MRd-CplD packet timing.

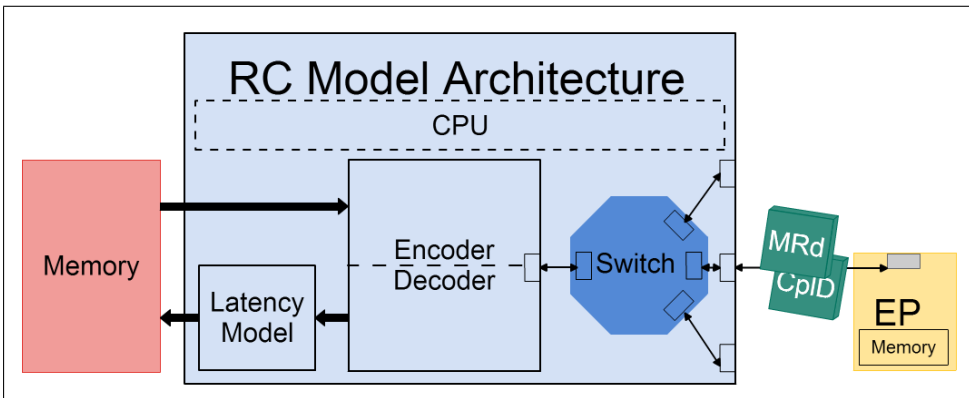
After creating a model of the PCIe RC, a suitable test environment was implemented for evaluation of the RC TLM. The environment consisted of an PCIe EP and switches. The EP model created in section 3.4 can be used as a blueprint for creating a TLM of the EP to be performance evaluated through simulations. Section 4.5 describes the verification process of the RC TLM.

## 3.1 Decision of Approach

### 3.1.1 Deciding on the Architectural Structure of the PCIe RC Model

Pre-studies of the QEMU software, and the PCIe system architecture user manual [3] were necessary to find a suitable approach towards modeling the entity that ties the PCIe system together with the CPU and memory. In addition, thoughts had to be given towards the user-friendliness and accuracy of the model structure. The RC model needs to be statistically accurate and to correctly test performance of PCIe EPs.

The RC emulator that is currently utilized by Oracle for performance measurements, the QEMU emulator, is a multi-purpose CPU-emulator. QEMU does not provide timing accuracy because of its structure as a processor emulator that runs on top of other hardware platforms. The emulated instructions are not executed on the target architecture, instead, they are simply mapped into local bare-metal instructions that consume hardware-dependent execution times. The QEMU emulator is described in more detail in section 2.3.



**Figure 3.2:** Structural overview of the RC TLM

A natural architecture of choice was found to be a software model that is interconnect-able with a functional-level software-model of the PCIe EP, the device under performance testing. The software model is illustrated in figure 3.2. The software model architecture was decided to be targeted solely towards its purpose, accuracy and intuitiveness of the model architecture were areas of focus. The RC model was designed to be able to build memory request packets on behalf of a simulated CPU and send them to connected devices, the resulting completion packets should be handled upon reception. The RC should also be able to receive incoming memory request packets and handle them either by simply writing to the simulated local subsystem memory or by reading from the simulated local subsystem memory and then build completion packets to return the resulting data.

To create a tool that is both simple, task specific and timing correct, it was decided that the model should be located in the PCIe transaction layer and in higher levels of abstraction. It should respond to requests after a delay that is statistically correct. The model architecture was chosen to have a pre-compile-time configuration, leaving out the complexity for setting up the system as well as support for hot-plug and similar functionalities. DLLPs, configuration packets, messages and IO transactions are not required for measurements of performance and was therefore not included in the architecture of choice, however they can be added later without any major changes in the software structure due to the modularity of the system.

SystemC was chosen as the modeling language because of its TLM abilities, further argumentation for the choice of modeling language are discussed in section 5.2.1.

### 3.1.2 Model Approach for Delay Correctness

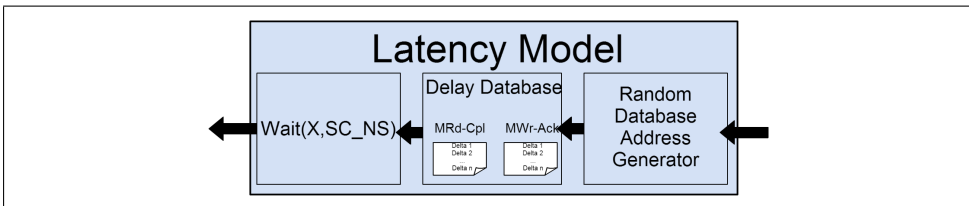
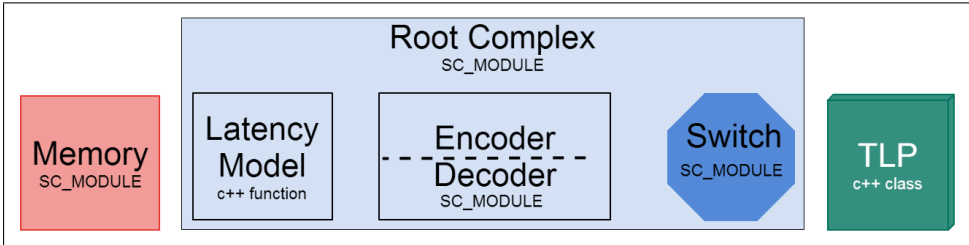


Figure 3.3: A realistic delay model

The delay-correctness of the RC model is ensured by using real hardware traffic traces from the Teledyne LeCroy tracer described in the theory section. Two model implementation variants of the RC delay-module were considered, both utilizing the wait statement in SystemC. The latency model that was chosen was the one that is statistically correct because it deploys the actual data from hardware traces for delay modeling. Request packets are answered with completion packets after a randomly drawn sample delay from a database of real hardware traces. The structure of the delay model is displayed in figure 3.3. The nature of the SystemC wait-statement makes the structure of the latency model quite simple.

## 3.2 Implementation of the RC model



**Figure 3.4:** Modules, classes and functions of the RC TLM

To model an RC with the system architecture given in figure 3.2, the following SystemC and C++ specific module that are listed in 3.4 are needed.

- A C++ class for the TLP.
- A SC\_MODULE socket used for packet-decoding/encoding.
- A SC\_MODULE switch, being a sub-module of the RC to enable multiple ports.
- A SC\_MODULE that represents the local system memory.
- And, finally a c++ function handling the modeled request delay.

Details of these modules are further explained below in the following subsections in illustrative class diagrams. The Code is also available at github [10] and can be reviewed there. For the class and module diagrams, their respective names are given at the top of each diagram, attributes are given in the middle field and methods are given at the bottom. The methods that are concurrent processes include a comment next to them with either SC\_THREAD, SC\_CTHREAD or SC\_METHOD. In addition, the modules that contain other submodules are also illustrated with submodules inside of them.

Generally, all classes and modules are implemented with respect to the “rule of three” principle in c++ programming [19]. It claims that if a class defines either a destructor, copy-constructor or an copy-assignment operator, then all three of them should be implemented. This is especially important when data-types include pointers to objects and when typecasting is performed. STL containers are used as frequently as possible for implementing buffers and queues. The code related to this thesis is written using Google’s coding style for c++ [34].

### 3.2.1 Implementing the TLP C++ class

The TLP class is implemented with a header-pointer, a data payload vector and an optional ECRC vector.

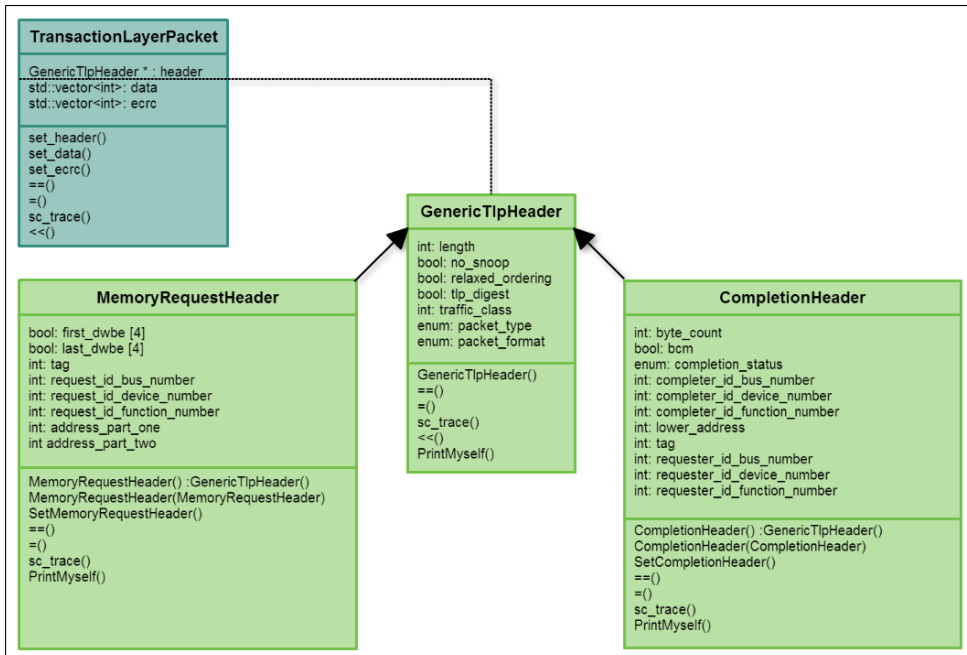


Figure 3.5: A class diagram of the TLP and its header structure

The five types of headers for TLPs; the IO requests, memory requests, configuration requests, completions and message requests introduced in section 2.1.2 have a set of attributes or header-fields in common. These parameters are added as member variables in a generic TLP header, a parent-struct. This is done together with the use of header-typecasting to have only one TLP - data-type that is transferred through SystemC ports. Instead of having 10 io-ports between PCIe-devices one can have two ports carrying TLPs with member pointers to the type `sc_in <GenericTlpHeader>` and `sc_out <GenericTlpHeader>`. Type-casting based on the generic type-field is done on reception of the TLP with the generic TLP-header on the other side of the communication-channel.

The TLP and its subclasses are implemented as structs with attributes and methods according to the the diagram in figure 3.5. Only the memory request header and the completion header are implemented because the other header-types are not of significance for performance simulations of EPs.

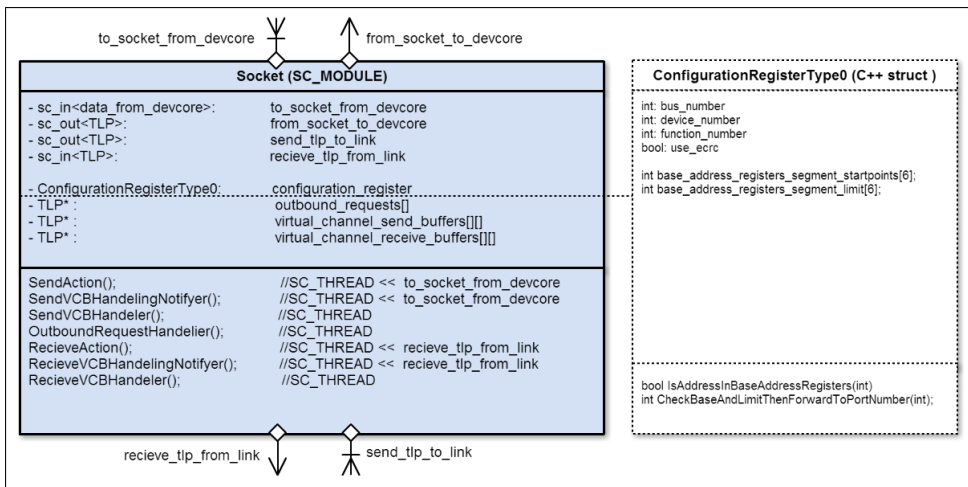
Note that the classes and structs in the diagram all have overloaded operators for equality checks, assignment operators, ostream operators and also overloaded `sc_trace()` functions. These are overloaded in order to support SystemC port-specific behavior.

The header child-types employ a workaround for inheriting the ostream operator since friend-functions can not be declared as virtual. Instead of overwriting the ostream function in the child data-types, the overloaded operator in the

parent type calls a virtual non-friend function called `PrintMyself` that is declared virtual.



### 3.2.2 Implementing the PCIe Encoder/Decoder Socket Module



**Figure 3.6:** A module diagram of the PCIe protocol stack

The PCIe Socket module is responsible for decoding incoming packets and for encoding packets to be sent that originates in the software layer of the corresponding device. The PCIe socket was made as a separate sub-module within the RC for reuse outside of the root-complex as a simple PCIe protocol stack for EPs. Figure 3.6 lists the most relevant members of the PCIe protocol decoder/encoder. The socket is the owner of the type 0 configuration register, containing information like BARs and device-IDs. The socket module is meant to be used as a submodule for basic EP and the RC. The socket was designed to have the 7 concurrent threads that are explained here.

- `SendAction()` thread is responsible for the entire process of encoding the data that arrives from the device core. This includes building the TLP based on the incoming data, adding ECRC and data. The packet is also added outbound request buffer if the software layer initiates a request. All packets that are built are sent to the virtual channel buffer corresponding to the traffic class of the packet for priority arbitration before the packet is sent.
- `SendVCBHandlingNotifier()` is a thread that assists to periodically notify the virtual channel buffer handler to check whether the queue is empty or not.
- `SendVCBHandler()` is the process that is responsible for performing priority buffer arbitration to empty the VCBs. Packets with a higher TC indicated in the header are prioritized for sending. And the buffer that is

emptied is collapsed for further handling of elements. Credits of VCBs are not taken into account here as the DLL is not part of the model. Buffers are assumed to be infinite.

- The `OutboundRequestHandler()` is responsible for surveiling all non-posted requests that are outbound, if a timer gives the handler a timeout notification, then the corresponding requests needs to be replayed.
- `RecieveAction()` listens to the PCIe channel and adds the incoming TLPs to the accompanying incoming VCBs for further handling.
- `RecieveVCBHandlingNotiflyer()` works in the same way as the notifier for the `sendVCB`, it periodically notifies the `receiveVCB` to perform arbitrations for handling the TLPs until the buffers are empty.
- `RecieveVCBHandeling()` performs arbitration of the incoming VCB, collapses the incoming VCB and then handles the TLP to be handled directly. The TLP is decoded, its optional ECRC field is checked for errors and is removed if the TLP is error free. The header and the data payload are then sent to the software layer on the `from_socket_to_devcore` port.

### 3.2.3 Implementing the PcieSwitch Module

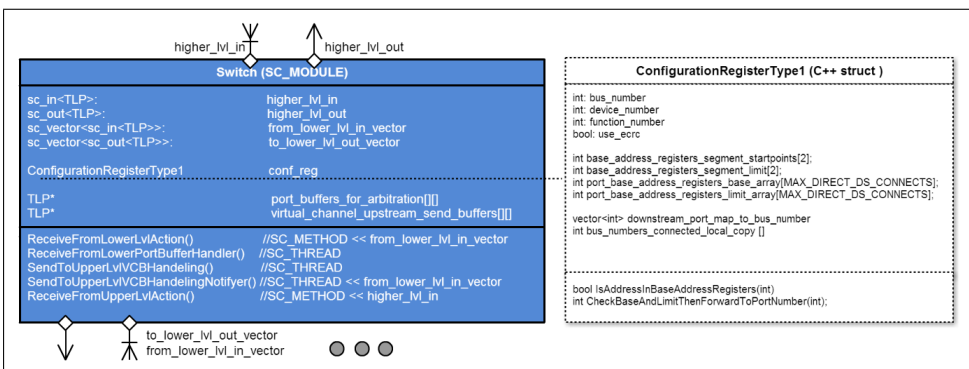


Figure 3.7: A module diagram of the PCIe packet switch

The PCIe switch located within the RC enables it to function as a multi-port device. The switch was made as a `SC_MODULE` for reuse outside of the root-complex as a simple PCIe packet switch for EPs. The module is the owner of a simplified version of the type 1 configuration register, containing information like BARs and deviceIDs in addition to system-memory information for each output port to allow address routing of packets flowing through the switch. A dynamic amount input and output ports are implemented to allow a more flexible test-setup, this is accomplished with the `sc_vector` datatype. The switch

was designed to have the 5 concurrent threads and methods that are explained in the list below.

- `ReceiveFromLowerLevelAction()` is the `SC_METHOD` that is run once for every delta-cycle when changes occurs on the vector of input ports from PCIe devices connected, that is when traffic is moving upstream. A simplification is done to the switch module, it is assumed that no PCIe EPs talk other PCIe devices than the RC. Hence all traffic arriving from a lower level can be forwarded directly upstreams towards the RC. Packets are allowed to arrive from multiple devices simultaneously per delta-cycle. This process inserts all the incoming packets into incoming port buffers.
- `ReceiveFromLowerPortBufferHandler()` handles the buffers that are filled up by the previously explained thread. Each packet arriving on the `from_lower_lvl_in_vector` is eventually inserted into a VCB for priority arbitration on the upstream port. First they have to be selected through input buffer arbitration, which is implemented with a fair round-robin selection scheme. Ports are in order allowed to forward their packets to the virtual channel buffer at the upstream port. A small delta-cycle delay is introduced for every packet that is routed from the input buffers to the egress port.
- `SendToUpperLvlVCBHandeling()` continuously tries to empty the buffer whenever packets are inserted into it, this is done with the same algorithm as the VCB handling threads in the Socket module.
- `SendToUpperLvlVCBHandlingNotifier()` periodically notifies the VCB handler to make sure that no packets are starved there.
- `RecieveFromUpperLvlAction()` works under the assumption that only one packet can arrive from a higher level of the switch per delta cycle. No VCB buffers or port buffers are needed as the packet can be directly forwarded to the correct port. The thread first checks the type-field in the generic packet header, if the header is a `CplD`, or a `Cpl` header, then the indicated Requester ID of the header is checked towards the Dev-ID to port number map located in the `typ1` configuration register. The packet is directly forward the packet to an egress port that is indicated in the ID-to-port-map. If either `MRd` or `MWr` is indicated, then the switch has to compare the address field located in the packet towards its arrays of base and limit addresses for each port. The packet is also directly forwarded during the same delta cycle as the arrival of the packet, because of the assumption that all from a downstream ingress port have to be forwarded upstreams, and the fact that there is only one upstream port.

### 3.2.4 Implementing the Subsystem Memory Module

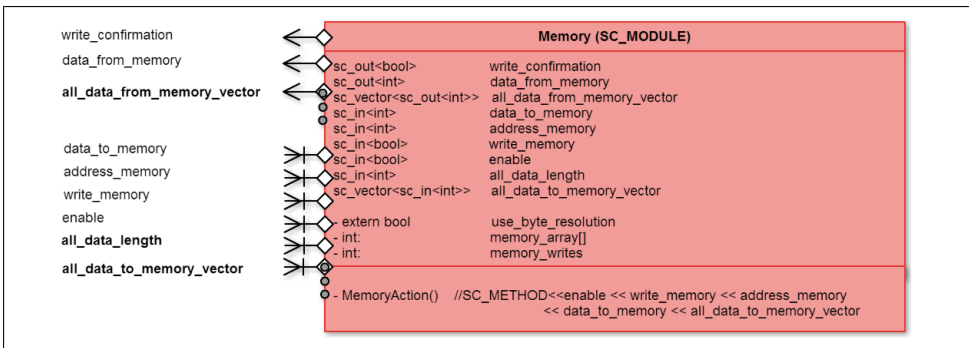


Figure 3.8: A module diagram of the host memory

The memory module is implemented as an array made out of integer values located within a SystemC module. The structure of the memory module is low level but also quite simple having only one process. The module is implemented to allow easy manipulation of the test-setup. One can set the the data resolution to either:

- Being byte-sized, returning one int that represents a byte for every delta cycle simulated.
- The entire amount of requested data can be either written or read during one delta cycle.

This setting is done by flipping the external global variable `use_byte_resolution`. The data buses on the module that are implemented using the `sc_vector<sc_in<int>` or the `sc_vector<sc_out<int>` template are enabled by default with the boolean being false. The byte resolution approach is set by setting the boolean high before commencing simulation. The bold iosignals in figure 3.8 are used to transfer the entire amount of requested data at once.

The `MemoryAction()` process is implemented as an `SC_METHOD`, it is executed once whenever there is action on either of the signals in the sensitivity list that is given in figure 3.8. The memory is manipulated in whatever manner is indicated by the inputs. If the `memory_write` bit is set, then the memory module will write memory, the data that is given on the input, and if the write bit is low, then the module will read the indicated data and return it to the RC. The `write_confirmation` bit is used during simulation of writing to memory during the byte-sized simulation of memory to let the RC know when it can send yet another write request. The `all_data_length` input is used for indicating the length of the read or the write whenever `use_byte_resolution` is low.

### 3.2.5 Implementing the RC Module.

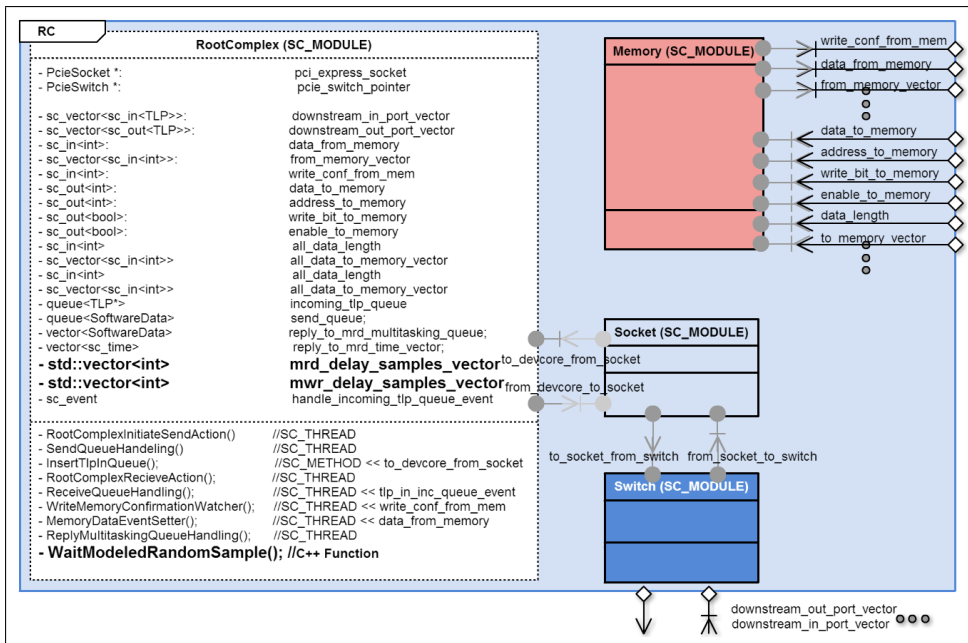


Figure 3.9: A module diagram of the RC TLM

RC attributes, io definitions, member functions as well as the memory interface of the RC module are shown in figure 3.9. The RC module consists of one instance of the Socket module and one instance of the switch module, the memory module is not per definition a sub-module, it is connected to the RC on a system-level. The Memory-module is connected to the RC's IO ports. That are marked on the outer edge of the RC. The bottom IO ports are connected directly to the switch sub-module. the switch might also be used outside of the RC module. Internal sub-modules are interconnected using SystemC signals (sc\_signals).

The socket within the RC is responsible for all actions related to building TLPs, decoding incoming TLPs, prioritizing sends, and keeping track of out-bound requests, hence these actions are not implemented directly in the RC, but indirectly through submodule-instantiation. In addition to including the functionalities of the submodules, the RC also consists of a large amount of member functions, the ones that are declared as processes are displayed in the figure and explained here:

- RootComplexInitiateSendAction() is implemented to periodically send a userdefined number of requests to the socket for building and sending to a completer. The function is implemented in a manner that is easy to edit

and modify to change the sending period, the amount of data to request, the request type and so on. The number of requests sent can also be 0. The data that is sent is a user-defined data format called `SoftwareData`, it is put into a queue for sending down to the socket module.

- `SendQueueHandling()` the queue for sending to the socket module is continuously handled by this thread, once every delta cycle to prevent clumping.
- `InsertTlpInQueue()` inserts any incoming TLP from the Socket module in the queue for handling, it also notifies the `RecieveQueueHandling` through a SystemC event.
- `RecieveQueueHandling()` continuously empties the incoming buffer whenever it gets filled up. An event is notified to tell the `RootComplexRecieveAction()` to handle the first element in the queue.
- `RootComplexRecieveAction()` reads the first packet from the incoming queue, the packet type is directly read from the generic header attribute “type” and a switch-case handles the packet differently for different packet-types. The resulting case dynamically casts the generic header located within the packet to the corresponding packet data-type. The corresponding action to the incoming request is performed.
- `WriteMemoryConfirmationWatcher()` lets the RC know when data has been written to the memory during the byte-resolution memory simulation.
- `ReplyMultitaskingQueueHandling()` is optional for the RC simulation, it enables multitasking. It checks the reply to mrd time vector every delta cycle, if a packet is to be sent during that cycle, then it sends the packet that corresponds to the location of the timestamp within the reply to mrd multitasking vector by checking. The two vectors are linked, each element corresponds to the element at the same address in the other vector.

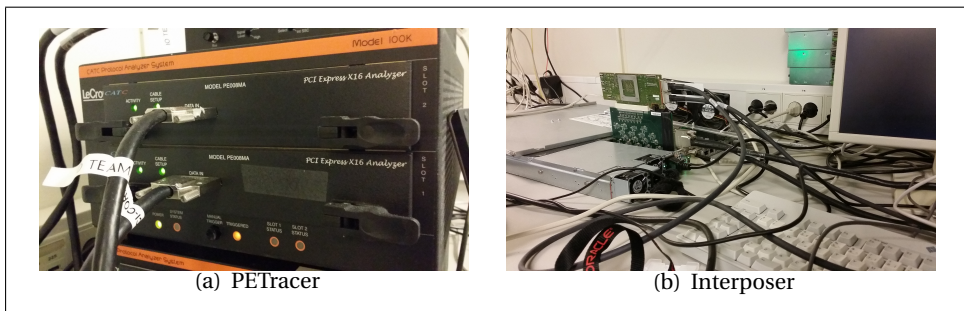
`WaitModeledRandomSample()` the final function that is listed in the figure in bold is the delay model of the RC. The model of the request-to-completion delay is imperative considering hardware-realistic accuracy of the RC model. The concept of the delay model function is quite simple. Whenever the `RecieveQueueHandling()` gets a non-posted request, the RC performs whatever action it is told to do by the request on one delta-cycle. And then, before sending the completion back to the requester, the RC calls `WaitModeledRandomSample()`. The function draws a delay sample from the `mrd_delay_samples_vector` that is filled up with delay samples on construction of the RC. The vector is marked with bold-phase in figure 3.9. The function then

waits for the time specified at the random location of the vector. Once the waiting is complete, the RC sends the completion packet back to the requester. In this way the completion will be returned after a statistically correct amount of simulated nano-seconds.

Optionally support for memory write modeling could also be implemented, delay data should be gathered from another approach than the Lecroy protocol analyzer, because writes are non-posted. Whenever the RC receives a posted request, such as a memory write request, the `WaitModeledRandomSample()` function is called in advance of performing the requested task. The function should be called on a different sample-vector, the `mwr_delay_samples_vector` once delay samples are available.

### 3.3 Performing traces on relevant HardWare

#### 3.3.1 Setting up the Trace Hardware and the Trace Software



**Figure 3.10:** Trace setup at Oracle's test lab

To generate text-files consisting of columns of delay-deltas, traces from real hardware had to be sampled and formatted. A setup including a PCIe gen 1 tracer was available at Oracle's office in Oslo, the equipment that was used for tracing the PCIe traffic is listed below and was connected in a similar manner to the setup shown in figure 2.31 in the theory section.

- Catc UPAS model 100k.
- 2x CATC, PCIe X16 Analyzer PE-EMLTracer (one for each data direction), Model PE008MA.
- PE16x InterPoser
- 16x Interposer Cables
- Workstation for controlling and recording traces, dell laptop with LeCroy PETracer' analyzer, version 5.71.
- PCIe Endpoint made by Oracle
- A SUN FIRE X4170 M2 Server having two Intel Xeon E5-2600 CPUs.[23][15]

The Catc Protocol analyzer system model 100k filled with two CATC Model PE008MA PCIe X16 Analyzer PE-EMLTracer - cards. A PCIe generation 1 card made by Oracle was docked into PCIe interposer system serving as the PCIe EP, details about the application running on it aswell as information about the card itself remains unrevealed to preserve the partial disclosure agreement with Oracle. The interposer was again docked into a PCIe slot in a SUN-server with a Xeon architecture. The two cables connected to the interposer were then connected to each of the PETracers located within the UPAS. The whole setup resembles



that of a normal ampere-meter, connecting serially into the PCIe-connection to sniff on the information flowing in the interconnect between the PCIe EP and the PCIe RC. The 100k is connected to an xp-machine via USB to transfer the wanted trace-data.

Only one set of trace-hardware was available at the time of tracing, that is why only PCIe-gen1 tracing was performed. Future completion of the RC-model requires tracing with PCIe gen-2 PCIe gen-3, depending on the architecture of the EPs to be performance-tested.

Setting	Status	Upstream settings :	Status
Recording Mode	Event trigger	Inhibit Channel :	No
Save As Multisegment Trace:	Disabled	Reverse Lines :	No
Buffer Size	768 MB	Invert Lane Polarity :	Autodetect
Post-trigger position	96%		
Use External Reference Clock :	Yes	<b>Downstream settings:</b>	
Base filename	data.pex	Inhibit Channel :	No
Save External Signals	No	Reverse Lines :	No
Use External Reference Clock :	Yes	Invert Lane Polarity :	Autodetect
Base Spec 1.0 Capability Mode :	No		
Descrambling :	On		
Link Width :	x16		

**Table 3.1:** The settings used for a trace recording

Table 3.1 shows the set-up extraction of the PETracer-software on the xp-machine. Event triggering is used, to enable recording once the desired event occurs on the PCIe bus. Buffer size is set to 768 MB this is the recording buffer size per direction. The Trigger position is set to 96% post-triggering which allocates 4% to pre-triggering recording and 96% to post-triggering recording. For this purpose 100% post-triggering might as well have been used, as the only purpose is to provide detailed information on read-requests, writes and completions which thanks to the application are quite a few and chronologically sparse located. The Trigger options are set to Memory Read and Memory Write requests to start recording once one of these packets occur. The recorded trace-data was stored in a pex file called data.pex. Link width for the tracing was set to 16x. No special settings were used for links or lane polarities and an external reference clock was used because of spread-spectrum clocking of the PCIe link under analysis.

### 3.3.2 Performing the Tracing

The command shown in figure 3.11 was used to execute a "RDMA Send/Receive ud\_bw UD streaming one way bandwidth" test. It is explained in detail in [39]. The application was executed on the EP to create a flow of packet traffic between the EP and the RC. The test is run in loopback mode, initiating DMA operations

```
qperf localhost -li sif0:1 -ri sif0:1 -lca 2 -rca 3 -vv -uu -mt 256 -m 256 -n 5 ←
ud_bw
```

**Figure 3.11:** OFED comand for executing an application on the PCIe EP

with MRd 32-bit from host memory, data that is recieved in completions are run in a loopback path in the EP, then it is written back to the RC and the host memmory using MWr32.

The application running on the EP used traffic class 0 only, to give a simple starting-point for the model. Recording of packets were started via the PEtracer SW. 5 GBs of traces were recorded over 3 trace iterations, providing trace-data from around 50 million packets whereas 16 million were TLPs. A high number of packets ensures that the delay distribution makes is correct due to the statistical law of many.

### 3.3.3 Converting the Trace Data to Text Delay Coloumns

The generated .pex files had to be converted into a readable format for the RC model.

Packet	R→	2.5	TLP	Mem	MRd(32)	Length	RequesterID	Tag	Address	1st BE	Last BE	LCRC	Time Delta	Time Stamp			
469489	R→	x16	3078	000:00000	32	160:00:0	31	FFFC5880	1111	1111	0xE3FD2B6	200.000 ns	0006_033 900 520 s				
469490	R→	x16	DLLP	ACK	AckNak_Seq_Num	3078	0x4258	Idle	44.000 ns	0006_033 900 720 s							
469491	R→	x16	SKIP	COM	SKIP Symbols	K28.0 K28.0 K28.0 K28.0	0.000 ns	0006_033 900 768 s									
469492	R→	x16	DLLP	UpdateFC-NP	VC ID	HdrFC	DataFC	CRC 16	Idle	84.000 ns	0006_033 900 784 s						
469493	R→	x16	TLP	Cpl	CplID	Length	RequesterID	Tag	CompleterID	Status	BCM	Byte Cnt	Lwr Addr	Data	LCRC	Time Delta	Time Stamp
469494	R→	x16	DLLP	ACK	AckNak_Seq_Num	3788	0x7555	2.552 us	0006_033 901 448 s								
469495	R→	x16	DLLP	UpdateFC-P	VC ID	HdrFC	DataFC	CRC 16	Idle	356.000 ns	0006_033 904 000 s						
469496	R→	x16	DLLP	UpdateFC-NP	VC ID	HdrFC	DataFC	CRC 16	Time Delta	368.000 ns	0006_033 904 360 s						
469497	R→	x16	SKIP	COM	SKIP Symbols	K28.0 K28.0 K28.0 K28.0	808.000 ns	0006_033 904 728 s									

**Figure 3.12:** Snippet of raw packet flow between the PCIe EP and the RC

Figure 3.12 shows a snapshot of one of the generated pex-files in the PE-Tracer software without any hiding of packets activated. The first packet is a TLP memory read request sent from the EP to the RC, followed by a DLLP who's purpose is to acknowledge the reception of the memory read request. The SKIP packet in purple is a symbol sequence used for compensation of different frequencies of bit-rate that a packet is transmitted with amongst different lanes. In a multi-lane environment SKIP sequences are transmitted simultaneously on all lanes. Packet number 469492 is a DLLP who's task is to update the credit-status of the RC's virtual channel buffers. The following packet is the one we are interested in, it is the completion of the memory read request. By subtracting its time-stamp from the time-stamp of its corresponding memory read request, we

can find the time it takes from a request sent by the EP until it receives a packet containing the data that it requested. All other packets in Figure 3.12 except from the MRd and the CplD can be ignored for MRd-Cpl latency extraction.

Packet	R-	2.5	TLP	Mem	MRd(32)	Length	RequesterID	Tag	Address	1st BE	Last BE	LCRC	Time Delta	Time Stamp			
469489	x16	3078	Mem	MRd(32)	32	160.00.0	31	FFFC5880	1111	1111	0xE3FDF2B6	352.000 ns	0006_033.900.520 s				
469493	R-	x16	3788	Cpl	CplD	32	160.00.0	31	000317	SC	0	128	0x00	32 dwords	0x7D02835B	65.456 us	0006_033.900.872 s
469560	R-	x16	3079	Mem	MWr(32)	8	160.00.0	0	FFFA1620	1111	1111	0 dwords	0xA7A9E44	10.464 us	0006_033.906.328 s		
469573	R-	x16	3080	Mem	MWr(32)	16	160.00.0	0	FFF1F3C0	1111	1111	16 dwords	0x012BF810	4.640 us	0006_033.976.792 s		
469579	R-	x16	3081	Mem	MWr(32)	1	160.00.0	0	FEE00218	1111	0000	1 dwords	0x0BB8EABA	84.864 us	0006_033.981.456 s		
469671	R-	x16	3789	Mem	MWr(64)	2	000.00.0	1	00003C1F.F8100040	1111	1111	2 dwords	0xB6C7075E	41.200 us	0006_034.066.320 s		
469715	R-	x16	3082	Mem	MRd(32)	32	160.00.0	30	FFFC5900	1111	1111	0xDC26B713	264.000 ns	0006_034.107.520 s			
469719	R-	x16	3790	Cpl	CplD	16	160.00.0	30	000.00.0	SC	0	128	0x00	16 dwords	0x3D81FF40	0.000 ns	0006_034.107.784 s
469720	R-	x16	3791	Cpl	CplD	16	160.00.0	30	000.00.0	SC	0	64	0x40	16 dwords	0x9A126F1D	17.984 us	0006_034.107.808 s

Figure 3.13: Snippet of TLP-only flow between the PCIe EP and the RC

Figure 3.13 shows the same view, only with the redundant packets removed. All other packets are hidden, this step was done to reduce the size of the text-files when exporting the data for further analysis. Text-files from the data-trace files are generated with the simple file->export->to\_text function.

The text box in figure 3.14 shows the exact same snippet of packets as in figure 3.13. Text is however a lot easier to analyze with self-made computer-tools. A program was written in c++ to read a text-file and extract the delta delays. The program is added to this thesis as appendix B. The delays are written back to a new text-file with one delta-delay given in nano seconds per row. An example of the final format is given in figure 3.15.

### 3.3.4 Linking the Delay Model and the Trace Data

Linking the data in the text-files together with the RC was a fairly simple task. The text-files were added directly to the folder containing the RC-model executable. The files were programmed to be opened RC constructor, casted to integer values and pushed to the delay-vector of the RC. Whenever a delay-sample is needed, the random-delay function of the RC generates a random address and withdraws its corresponding delay for use in the wait statement.

```

Pkt(469489) Upstream TLP(3078) Mem MRd(32)(000:00000) MRd(32)(000:00000)
-----| RequesterID(160:00:0) Tag(31) Address(FFFC5880) LCRC(0xE3FDF2B6)
-----| Time Stamp(0006 . 033 900 520 s) UnkwnField
-----|
Pkt(469493) Downstream TLP(3788) Cpl CplD(010:01010) CplD(010:01010)
-----| RequesterID(160:00:0) Tag(31) CompleterID(000:31:7) Data(32 dwords)
-----| LCRC(0x7D02835B) Time Stamp(0006 . 033 900 872 s) UnkwnField
-----|
Pkt(469560) Upstream TLP(3079) Mem MWr(32)(010:00000) MWr(32)(010:00000)
-----| RequesterID(160:00:0) Tag(0) Address(FFFA1620) Data(8 dwords)
-----| LCRC(0xA7AA9E44) Time Stamp(0006 . 033 966 328 s) UnkwnField
-----|
Pkt(469573) Upstream TLP(3080) Mem MWr(32)(010:00000) MWr(32)(010:00000)
-----| RequesterID(160:00:0) Tag(0) Address(FFF1F3C0) Data(16 dwords)
-----| LCRC(0x012BFB10) Time Stamp(0006 . 033 976 792 s) UnkwnField
-----|
Pkt(469579) Upstream TLP(3081) Mem MWr(32)(010:00000) MWr(32)(010:00000)
-----| RequesterID(160:00:0) Tag(0) Address(FEE00218) Data(1 dword)
-----| LCRC(0x0BBEABA) Time Stamp(0006 . 033 981 456 s) UnkwnField
-----|
Pkt(469671) Downstream TLP(3789) Mem MWr(64)(011:00000) MWr(64)(011:00000)
-----| RequesterID(000:00:0) Tag(1) Address(00003C1F:F8100040) Data(2 dwords)
-----| LCRC(0xB6C7075E) Time Stamp(0006 . 034 066 320 s) UnkwnField
-----|
Pkt(469715) Upstream TLP(3082) Mem MRd(32)(000:00000) MRd(32)(000:00000)
-----| RequesterID(160:00:0) Tag(30) Address(FFFC5900) LCRC(0xDC25B713)
-----| Time Stamp(0006 . 034 107 520 s) UnkwnField
-----|
Pkt(469719) Downstream TLP(3790) Cpl CplD(010:01010) CplD(010:01010)
-----| RequesterID(160:00:0) Tag(30) CompleterID(000:00:0) Data(16 dwords)
-----| LCRC(0x3D81FF40) Time Stamp(0006 . 034 107 784 s) UnkwnField
-----|
Pkt(469720) Downstream TLP(3791) Cpl CplD(010:01010) CplD(010:01010)
-----| RequesterID(160:00:0) Tag(30) CompleterID(000:00:0) Data(16 dwords)
-----| LCRC(0x9A126F1D) Time Stamp(0006 . 034 107 808 s) UnkwnField
-----|

```

**Figure 3.14:** Example of TLP Traffic extracted to Text

```

<mrd-cpl delta 1>
<mrd-cpl delta 2>
<mrd-cpl delta 3>
...
...
...
<mrd-cpl delta n>

```

**Figure 3.15:** The Text Format Required by the RC Delay Model

### 3.4 Creating a test-environment for the RC

The functionality of the SystemC modeled RC had to be tested. This requires a system to do performance-measurements on. Because no design-stage EP model was provided, a SystemC model of an EP also had to be created to provide a realistic test-bench of the software. In this section the construction of a flexible PCIe EP module is described. The EP is combined together with the other modules that are created. The interconnecting process of a complete PCIe system, including memory, RC, switches and EPs is described in this section. Modules are interconnected with the `sc_signal` data type within a complete system module.

#### 3.4.1 Constructing the PCIe endpoint model

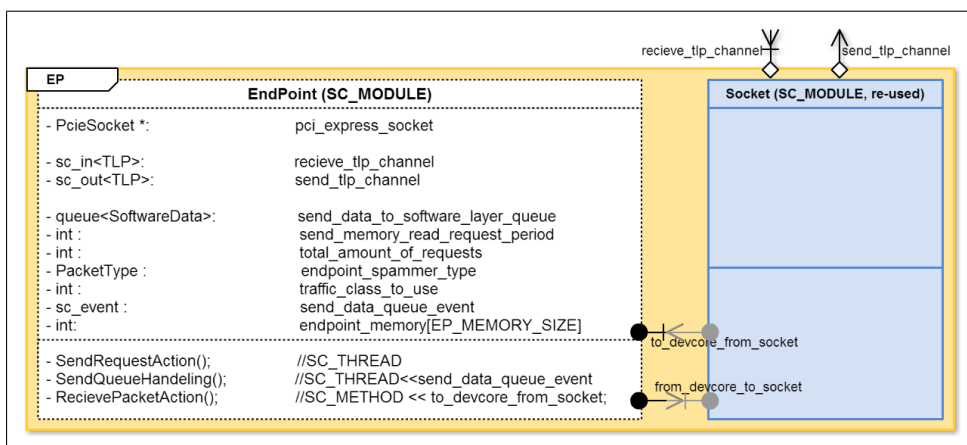


Figure 3.16: A module diagram of the PCIe EP TLM

Figure 3.16 shows the module structure of the PCIe EP module with an internal instance of the PCIe socket module. The functions and attributes that are listed are only a small fraction of the total module. The EP is designed as a SystemC module to allow multiple simultaneous EP instances for testing of the RC functionality in extreme environments. The module was designed to send a burst if requests to the RC, and to wait for it's replies. The EP module is also designed to be able to answer requests that originates in the RC. The request details and the type 0 EP configuration registers are easily set through the module constructor.

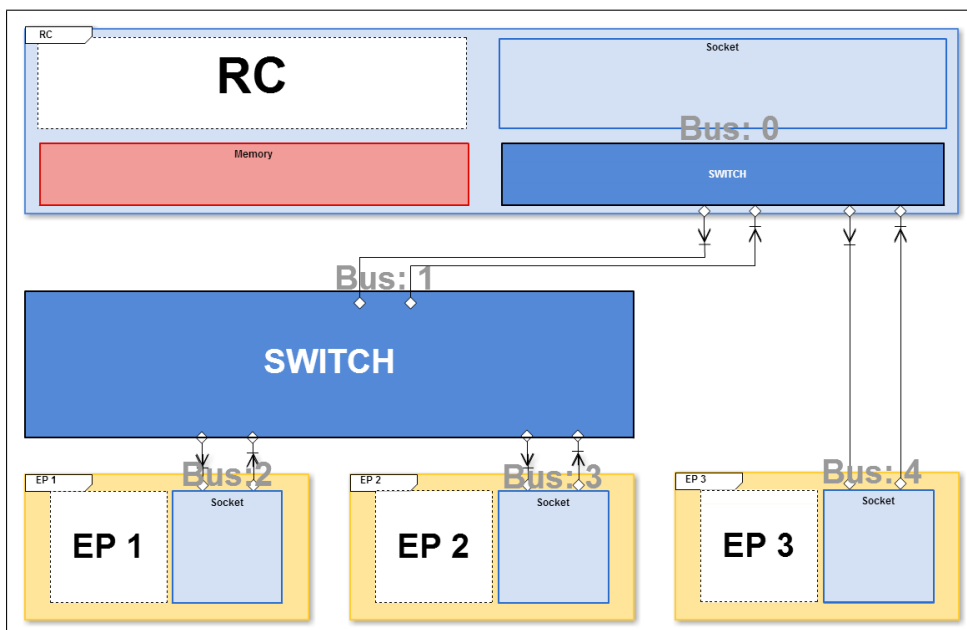
The endpoint module is built using three concurrent processes:

- `SendRequestAction()` is implemented as a `SC_THREAD`, it was designed to be responsible for sending request data down to the socket module for further packet construction and forwarding to the RC. The functionality

of this process was manipulated a couple of times during the RC model testing to provide different request scenarios to the RC. The process adds all requests to the internal software-layer send queue for further forwarding to the socket.

- `SendQueueHandling()` is designed as a `SC_THREAD`. The process continuously sends a data load to the socket every delta-cycle, whenever there is data to be sent. If the data to be sent is a CplD packet that originates from a MRd request, then an arbitrary delay can be set.
- `RecievePacketAction()` is a `SC_METHOD` that is run whenever packets arrive from the socket, similar to the receive function of the RC. The method answers request packets by reading from the internally simplified memory array and adding a completion packet to the send queue.

### 3.4.2 Interconnecting modules creating a complete system



**Figure 3.17:** A module diagram of the complete test system

A complete system including 3 EPs, one switch and an instance of the PCIe RC makes up the test set-up for the RC. The modules were instantiated within an outer shell SystemC-module. The complete system was interconnected as shown in figure 3.17. EP 3 is connected directly to one of the root-complex's multi-ports while EP 1 and EP 2 are connected to a switch that switches packets upstream towards the RC. Having multiple devices connected to the RC, allows

testing of scenarios with multiple requesters. When several devices simultaneously sends requests, a few delta-cycles of extra packet delay is introduced due to switching and buffer arbitration. This is not the case when only one device is requesting as the packet is allowed full bus allocation, the only delay that is experienced is the modeled delay in the RC. The system shown in the figure provides a flexible model without having to change the code for every different test scenario. However the system can be easily modified within the `complete_system.h` file by deleting or instantiating modules.

### 3.4.3 Creating the Program Interface in `sc_main.cpp`

The complete system is instantiated in `sc_main.cpp` before execution of the simulation kernel. The program itself is called from the terminal with settings given as command-line options. The flags that the programs are executed with are analyzed and used to set global variables describing EP and RC behavior through constructors. By executing the simulation program together with its parameters instead of taking inputs in runtime, the program is easily piped to unix functionalities such as `grep`. Having command line arguments also eases the use memory check tools such as `Valgrind`.

A dynamic setup of the program using extern global variables in together with command line arguments, allows the tester to control system variables. The global variables are used within the constructors of the system modules, as well as directly to describe output-options. Examples of variables that are controlled using the command line options are:

- The number of requests that each of the PCIe modules initiates.
- Type of requests that each of the PCIe modules initiates.
- The period between each request for all the PCIe modules given in nanoseconds. Having a lower request-period than the highest delay sample might lead to packet clumping. Sending packets b2b ideal for testing the clumping of the EPs.
- The total SystemC simulation time.
- The traffic class that each module delivers requests with. This feature allows the tester to check priority algorithms within the PCIe devices.

The `sc_main.cpp` file is designed with these variables to make the testing-job easier.

The program was designed to be executed with a format such as the snippet in figure 3.18. The first 6 command-line parameters are dedicated to setting the number of packets to be sent, together with the type of packets to be sent

```
./PCIESystem.out [#EP1-requests][#EP2-requests][#EP3-requests][EPs-req-type] [#RC-requests][RC-req-type] [Request-size-all][Simulation-time-ns] [Output-detail-flag][Write-output-to-textfile-flag][plot-traced-vs-RCsimulated-vs-EPexperienced-latency-distribution-flag]
```

**Figure 3.18:** The command-line syntax for executing a simulation

from each PCIe device in the system. The simulation time to execute the simulation kernel with is with the 7th argument. The final parameters are dedicated to visualizing the output of the simulation. The view-parameter allows the simulation output to be summarized and increased with respect to details. The write to console parameter allows the simulation output to be written to a text file. Finally the the plot option runs a python script in the terminal that plots the logged memory read to completion delays that are simulated by the complete system. Many more options are available for setting up the system, to enable these as command-line arguments, the program has to be slightly modified and the system then needs to be recompiled.

## 3.5 Testing the PCIe RC Model

The system was implemented using a bottom up approach, hence the submodules have been tested once they have been completed. The focus of this testing section is functionality testing of the RC module as a system.

### 3.5.1 System functionality

The packet history in each simulation is monitored by all the individual modules using counters and SystemC time-stamp buffers. The data stored during simulation is visualized in various ways in the terminal and with plots when the modules are destructed. The representation of the packet history depends on system settings at program execution time.

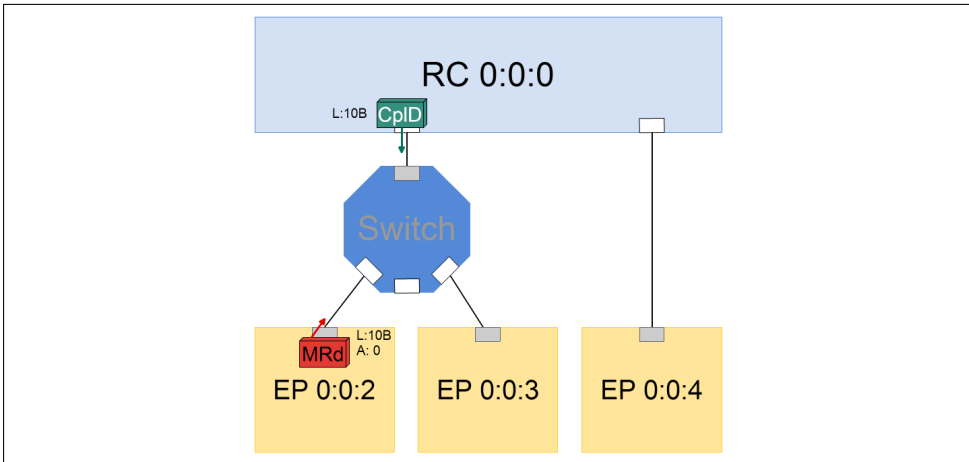


**Test-scenarios for the RC**

Multiple test scenarios were proposed and implemented in the software layers of the RC and EP modules. The following tests were conducted on the system to test the basic functionality of the RC:

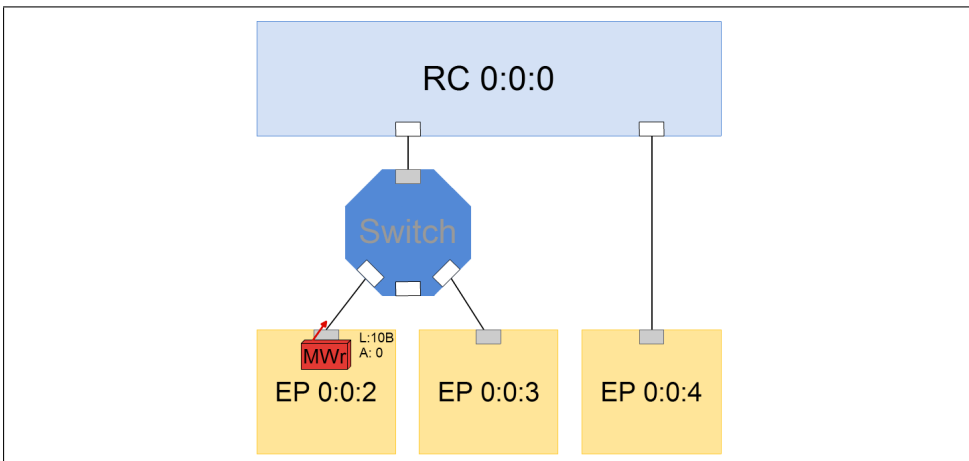
### The Root Complex as a completer:

- Answering single Requests per simulation:
  - Answering a MRd with a CplID that contains memory data from the system memory located locally with the RC.



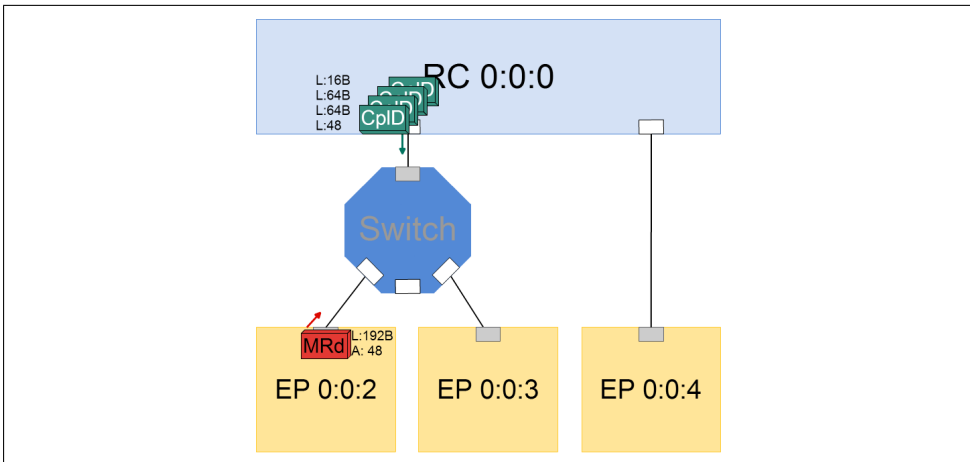
**Figure 3.19:** Testing a single MRd from EP:002 to the RC

- Answering a MWr by writing data to the system memory located locally with the RC. No packets are returned as Ack-Nak is a DLLP.



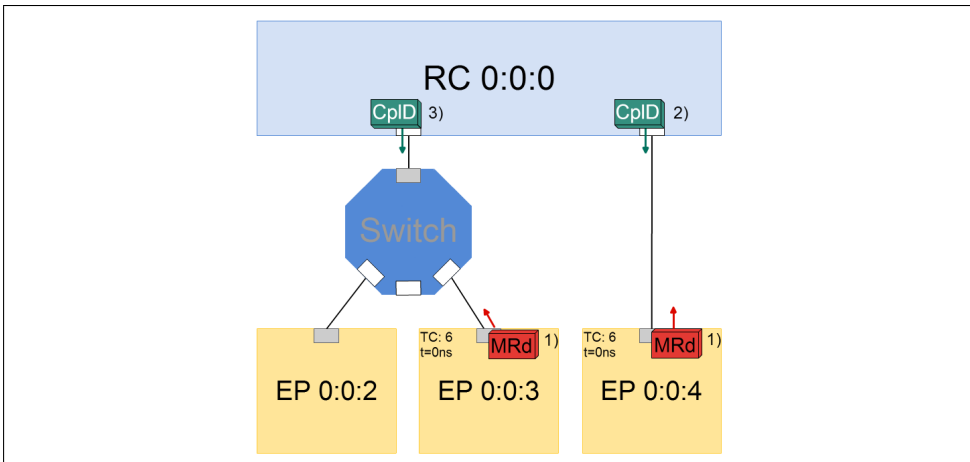
**Figure 3.20:** Testing a single MWr from EP:002 to the RC's system memory

- Answering a large MRd with multiple CplD due to the RCB. The example from the theory section.



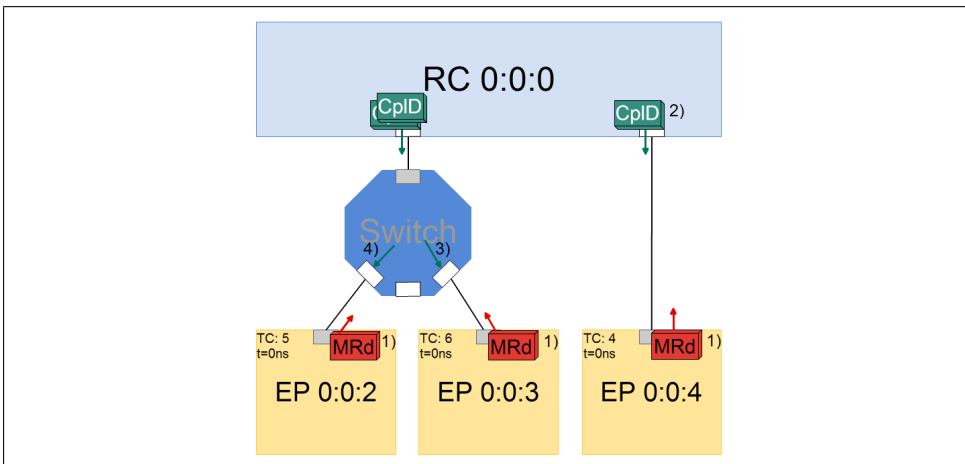
**Figure 3.21:** Testing a single MRd from EP:002 to the RC, RCB exceeded

- Answering multiple Requests per simulation:
  - Answering 2 requests, one from the EP behind the switch and one from the EP that is directly connected to the RC. The two requests are sent simultaneously and has the same traffic class. The small delay in the switch will influence the vcb receive handling.



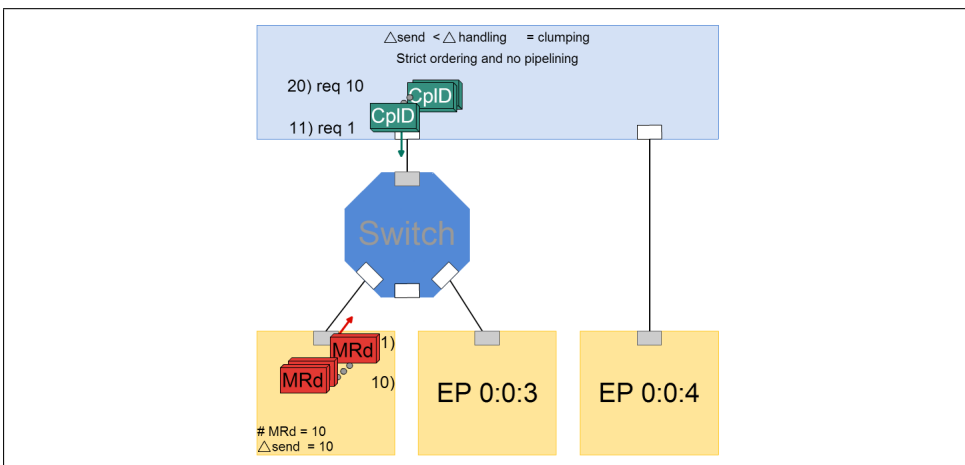
**Figure 3.22:** Testing 2 simultaneous requests, switching cycles are illustrated

- Answering a total of 3 MRd sent simultaneously from the 3 EPs. All EPs were configured to send packets with different traffic classes. This tests the packet priority arbitration of virtual channel buffers. The delta delay of the switch is also expected to have an impact on the arrival time of the requests.



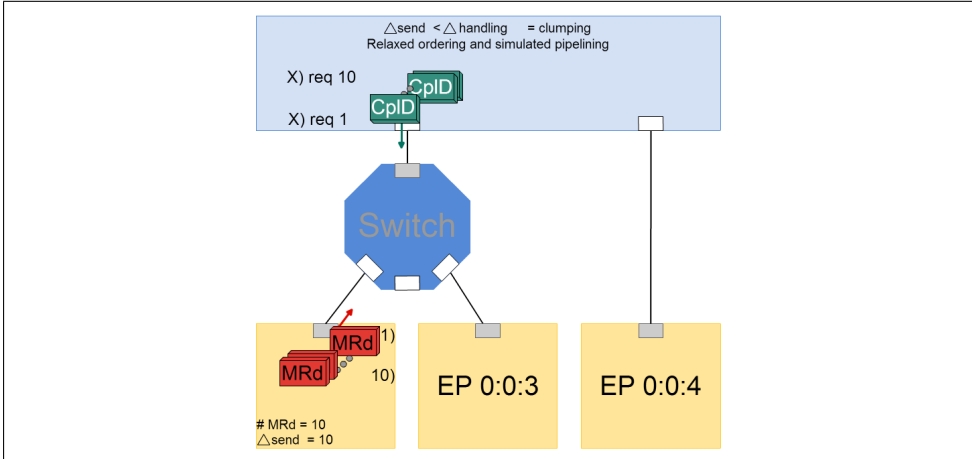
**Figure 3.23:** Testing the virtual channel buffer arbitration

- Answering multiple B2B requests to find Clumping vulnerabilities upstream. Strict ordering is used along with no pipelining of requests meaning that average MRd-cpl delay will add up for every additional packet sent, and the packets will be returned in order.



**Figure 3.24:** Testing rapidly sent requests

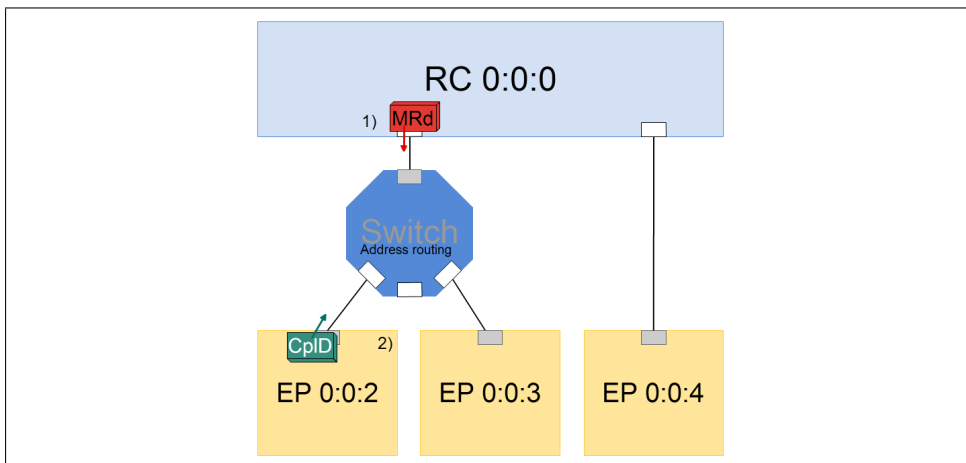
- Answering multiple B2B requests to in a similar fashion to the test approach described in the previous approach, however the ordering is relaxed and pipelining is ideally simulated the average MRd-CplD delay is expected to remain unchanged from single requests-completions.



**Figure 3.25:** Testing rapidly sent requests, RC relaxed ordering

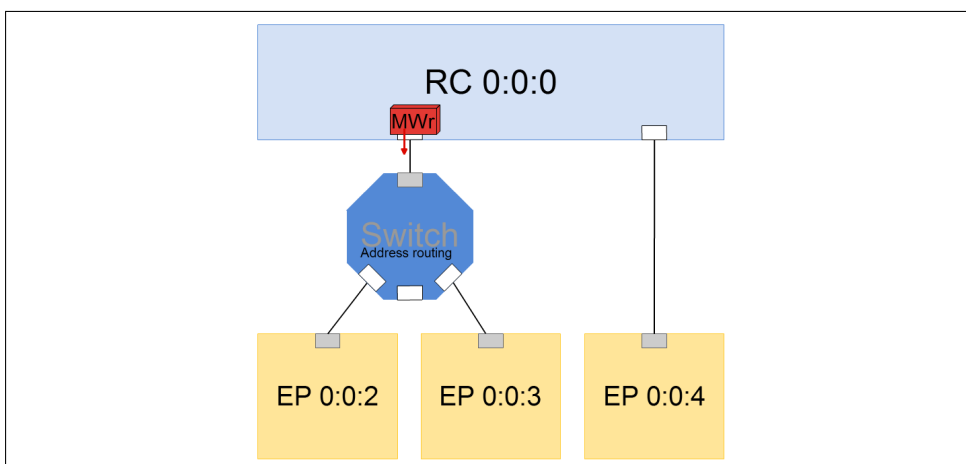
### The Root Complex as a Requester:

- Sending a single request per simulation:
  - 1 MRd TLP was sent with a memory address that resides within EP 0:0:2. Address routing is tested downstream. Correctness of the basic system memory read functionality was ensured by checking that the RC actually receives a packet containing the data that it requested from system memory. All fields in the packet-header were checked for correctness.



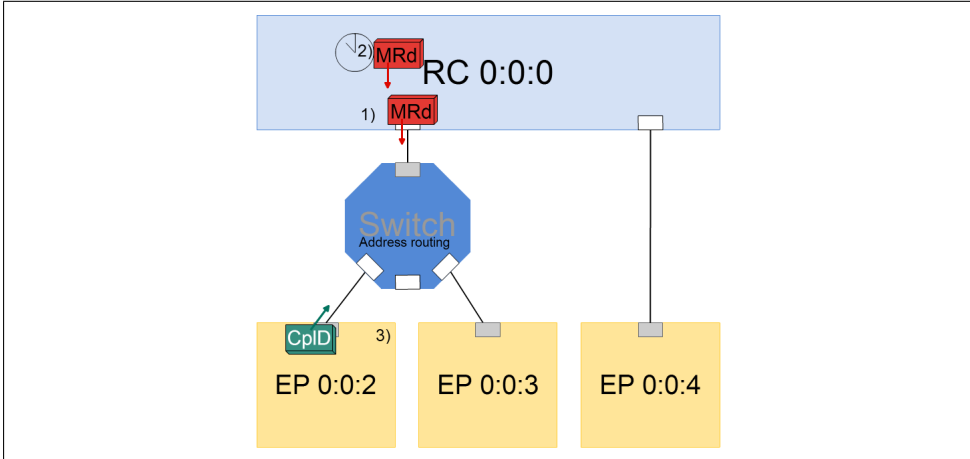
**Figure 3.26:** Testing RC's ability to issue requests, in this case a simple MRd

- 1 MWr TLP, correctness of the basic system memory write functionality was ensured through observation of the subsystem memory-location. The packet had to make its way all the way through the PCIe system fabric based on the location of the system memory address.



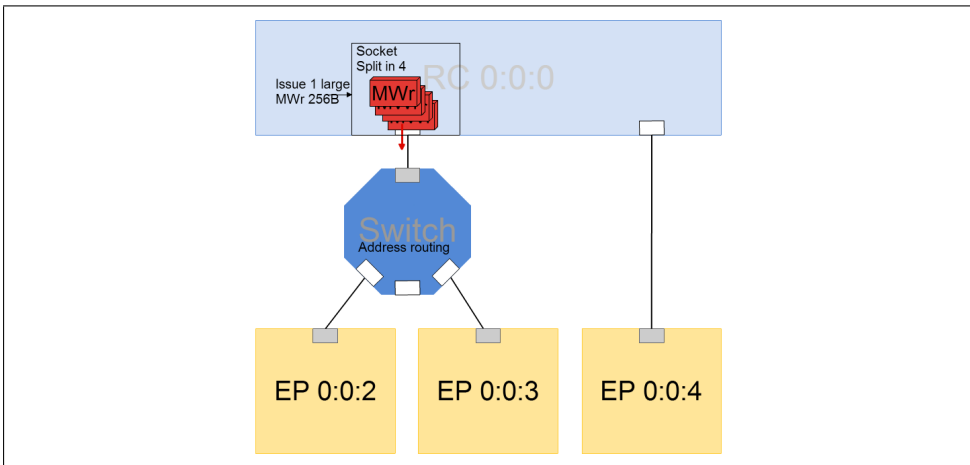
**Figure 3.27:** Testing RC's ability to issue a simple MWr request

- 1 request that times out, tests the replay functionality in the socket module. The EP module is reprogrammed to simply loose a package. The outbound buffer's replay functionality is tested within the RC, a timeout should be received issuing a new request.



**Figure 3.28:** Testing the RC's ability renew lost requests

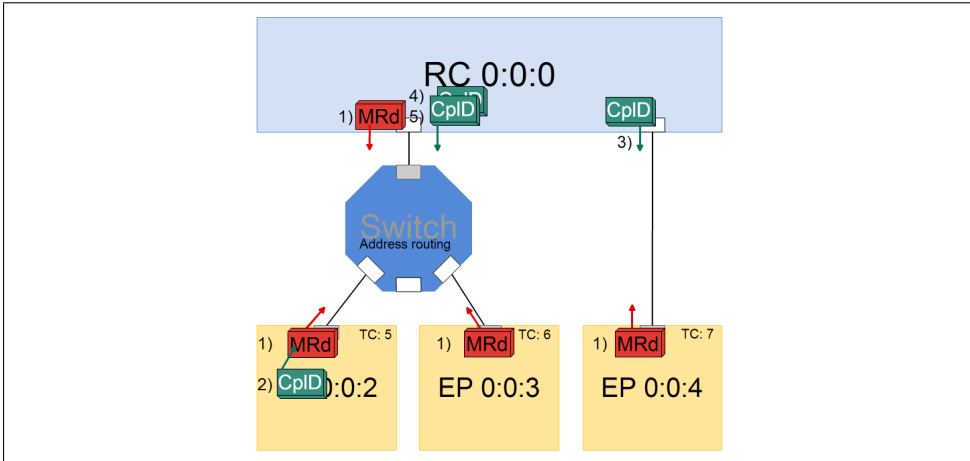
- 1 MWr with a large payload is issued to to the transaction layer of the RC. The RC's ability to divide the single request into multiple request due to the Max\_Payload\_size restriction is tested.



**Figure 3.29:** Testing the maximum payload size restriction with a memory write

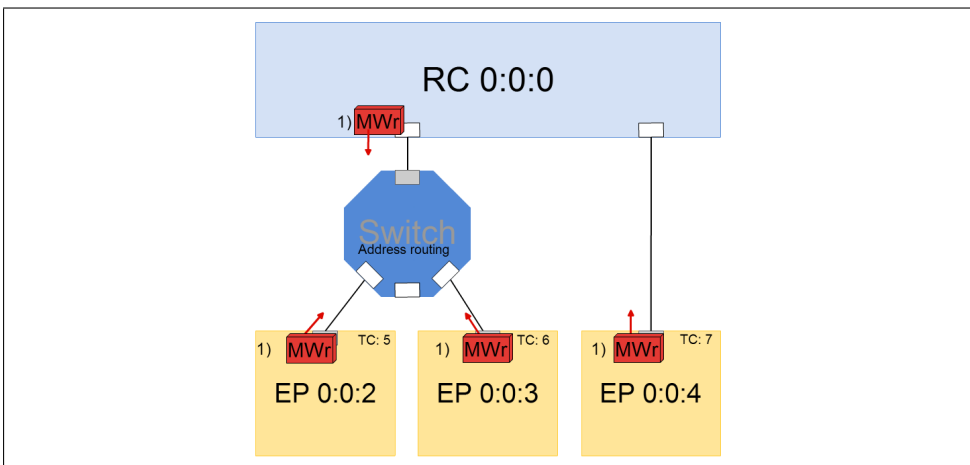
### Every PCIe device requests:

- 1 MRd was sent from every EP and from the RC simultaneously. Bidirectional PCIe read requests are tested.



**Figure 3.30:** Testing bidirectional MRd traffic

- 1 MWr was sent from every EP and from the RC simultaneously. Bidirectional PCIe write requests are tested.

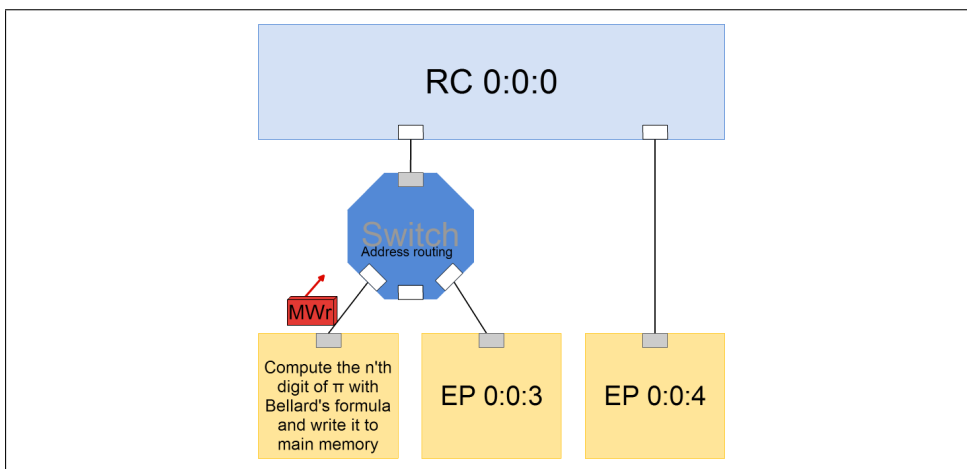


**Figure 3.31:** Testing bidirectional MWr traffic



### A Realistic Test-Scenario

- The entire system at focus, the  $n$ 'th digit of  $\pi$  is calculated on the  $n$ 'th delta cycle and is written to the RC memory every cycle. A realistic traffic scenario was also simulated on the system as a whole. Computational tasks are outsourced to the EP, the EP continuously calculates the  $n$ th digit of  $\pi$  using Bellard's algorithm and writes this digit to the RC system memory. This makes sense because calculating the  $n$ 'th digit of  $\pi$  with Bellard's formula takes  $O(n^2)$  time, it scales at the decimal point moves further and further to the left. The CPU will be able to continue with other tasks as its PCIe EP continuously writes a new digit of  $\pi$  to the RC subsystem memory once it is calculated. The simulation was done for 900 decimals of  $\pi$  that were sequentially written to the memory. To simplify the example, the computational time of  $\pi$  is simulated to be constant.



**Figure 3.32:** A realistic PCIe scenario, computation is outsourced to the EP

During all the above tests, the complete system was indirectly tested for correctness on displaying packet traffic statistics.

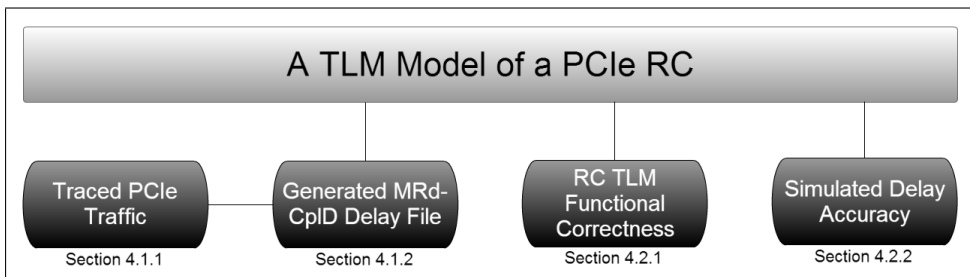
### 3.5.2 Valgrind Memory Leak Check for Runability

The complete system was debugged with the Valgrind memory check tool. The Valgrind check was performed in order to ensure error-free simulation in the long-run. Performance testings are often done over a long period of time, a small leak might cause a test-server to run out of memory and crash.

```
valgrind --leak-check=full --log-file="valgrind-log" [./PCIeSystem.out [normal←
args]]
```

**Figure 3.33:** The command-line syntax for a Valgrind execution

The program was run with various options testing the different modules together with the Valgrind command shown in figure [3.33](#):



**Figure 4.1:** Overview of the result sections in this chapter

The results discovered in this project are described in this chapter in a chronological manner. Data extracted from trace summaries of the LeCroy PCIe Tracings are given in subsection 4.1.1. The following subsection illustrates the extracted MRd-CplD latencies with graphs of latency distributions. Results proving the functional correctness of the RC TLM are shown in the form of simulation outputs in section 4.2.1. The final section in this chapter includes comparison graphs between simulated and sampled latency distributions to prove the TLM's latency accuracy.

## 4.1 Packet Tracing

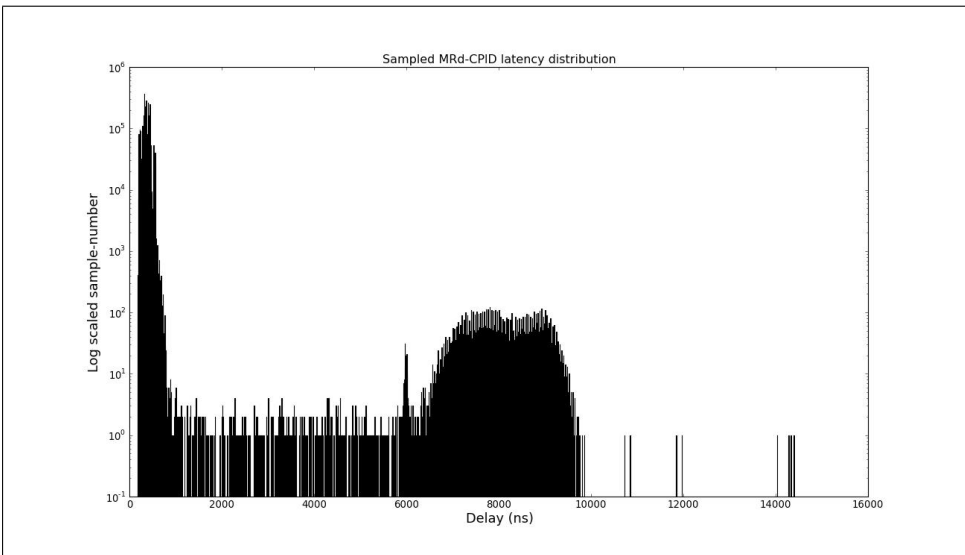
### 4.1.1 The PETracer Recordings

The results from the 3 iterations of traffic-recording were summarized in the PETracer software as html files. These were exported to PDFs and attached to appendix D. The total amount of traffic for the different TLP types of the 3 trace iterations of the same EP-application are shown in table 4.1.

Packet Type	Upstream	Downstream:	Total
Memory Read (32 bit)	5812015		5812015
Memory Write (32 bit)	3153216		3153216
Memory Write (64 bit)		418303	418303
Completion with Data		6657073	6657073
	8965231	7075376	16040607

**Table 4.1:** TLP-type trace statistics summary for all 3 trace iterations

## 4.1.2 The Generated Delay File for the RC Model



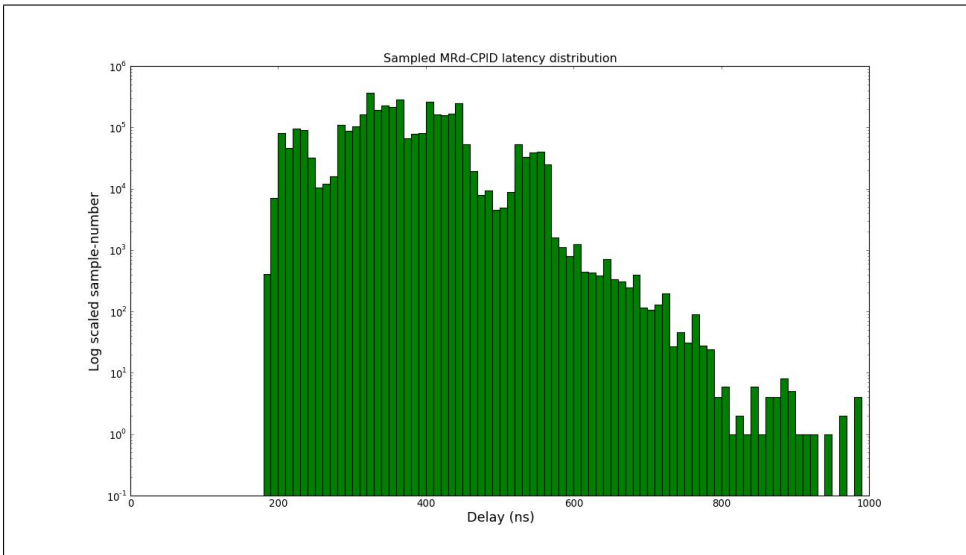
**Figure 4.2:** A histogram of the total amount of traced delays

The graph in figure 4.2 is a histogram of the MRd-CplD delays that were extracted from text trace files such as those shown in the methodology chapter in figure 3.14 using the C++ script in appendix B. The plot represents the MRd-CplD latency distribution of the PCIe RC in the SUN FIRE X4170 M3 server. It is important to note that all traffic used to generate this plot had the same packet priority, used strict ordering and snooping. The plot is generated using the same script as the one used in the PCIe System Model, using Python’s matplotlib library.

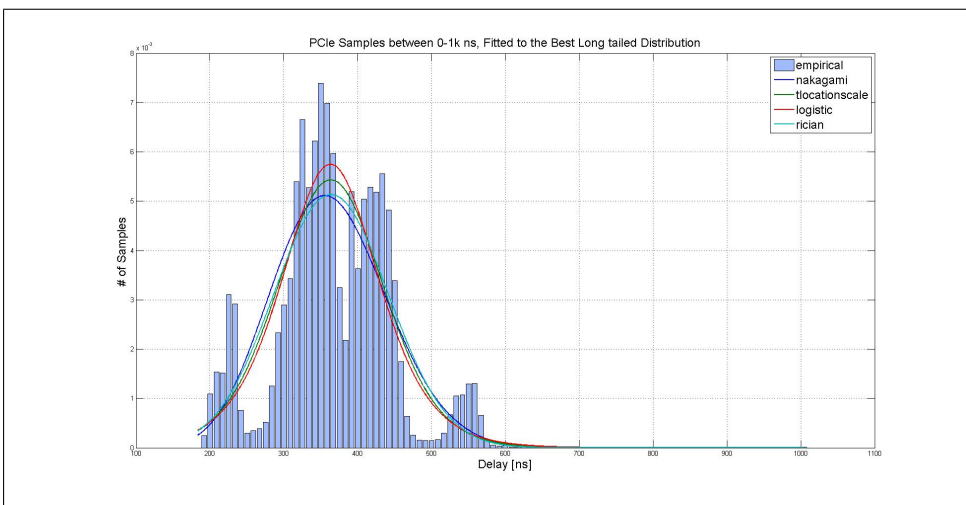
Metric	Delay [ns]
Minimum MRd-CplD delay	184 ns
Maximum MRd-CplD delay	15768 ns
Average MRd-CplD delay	393 ns

**Table 4.2:** General statistics for the recorded traces

Table 4.2 describes the key values of the generated delay file. The minimum, maximum and the average MRd-CplD latency for the traces are given in nano seconds.



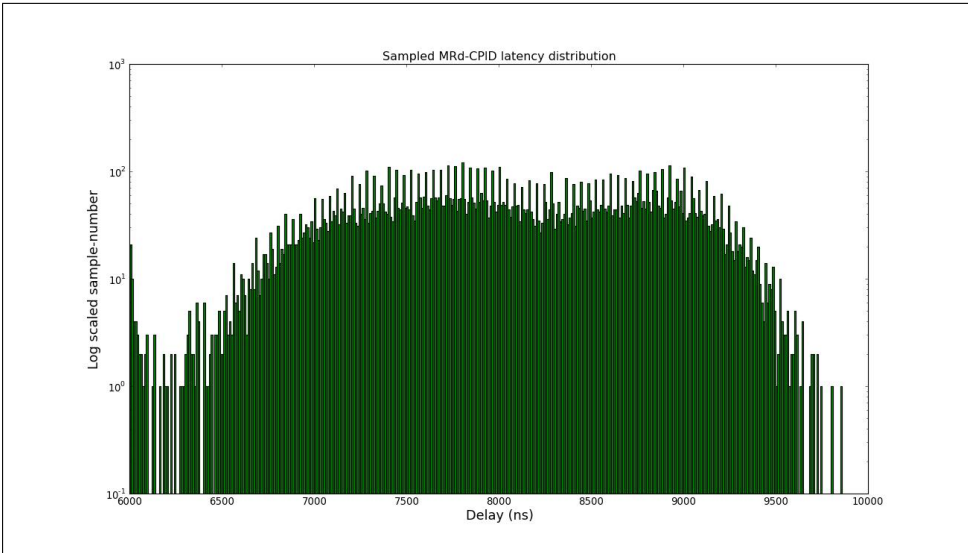
**Figure 4.3:** The first sub distribution in figure 4.2, heavy-tailed



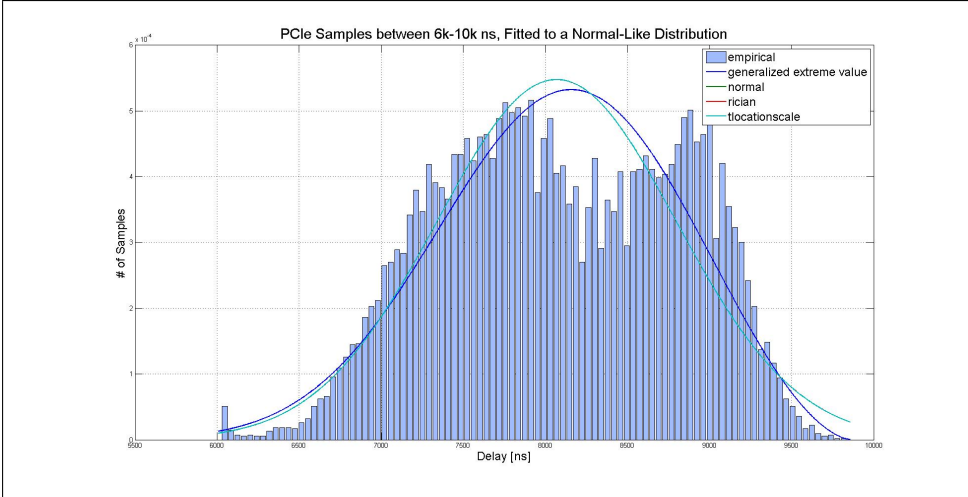
**Figure 4.4:** Fitting a probability distribution to the first sub distribution

The graph in figure 4.3 is a histogram of the MRd-CplD delay file plotting the first group/collection of delays. This group is the most heavily weighted one with 3659394 MRd-CplD combinations. It can be seen from the graph that this delay collection can be modeled using a long tailed probability distribution.

Matlab was used to find the most appropriate probability distribution to model this section, this is shown in figure 4.4.



**Figure 4.5:** The last sub distribution in figure 4.2, bell shape



**Figure 4.6:** Fitting a probability distribution to the last sub distribution

The graph in figure 4.5 is a histogram of the MRd-CplD delay file plotting the latter group/collecion of delays. The graph structure resembles that of a bell-curve with a mean of around 8000ns. This group is the lease heavily weighted one with only 13696 MRd-CplD combinations between 6000ns and 10000 ns. Matlab was used to find the most appropriate probability distribution to model

this section of the total latency distribution, this is shown in figure [4.6](#).

## 4.2 Functionality Test-Results of the RC

The simulation summaries displayed in this section were generated by logging terminal outputs of simulations using the C++ `freopen` command. The plots are automatically plotted using systemcalls to run python's `matplotlib` on the log-files.

### 4.2.1 Functional Accuracy

This section reveals the results from the system tests that were conducted and introduced in section [3.5](#). Program outputs were converted to text and the text files were snipped, summarize them for readability and added here. For an example of a full simulation output text file see the single MRd-CplD example that is attached in appendix [E.1](#). All other examples are summarized with the destructor outputs only. Special features are turned off unless otherwise stated.

## The Simulation Results for the RC Acting as a Completer

- Answering single Requests per simulation:
  - Answering a MRd with a CplD that contains memory data from RC's system memory fraction:

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
  ./PCIESystem.out 1 0 0 MRd 0 MRd 10000 -show_full -out.txt
Simulation statistics:
  Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
  Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
  Packet Statistics: <0/0> <1/1>
  Requests received time stamps:
    6 ns
  Completions sent time stamps:
    328 ns
--Switch: 000 Destructor--
  Packet Statistics: <1/1> <1/1>
--Socket Destructor--
  Packet Statistics: <1/1>

--EndPoint: 200 Destructor--
  Packet Statistics: <1/1> <0/0>
  Requests sent time stamps:
    0 s
  Completions received time stamps:
    328 ns
  Average MRd-CplD time is:
    328000 ps
  Data received is:
    1 2 3 4 5
    6 7 8 9 10
--Socket Destructor--
  Packet Statistics: <1/1>

--Switch: 100 Destructor--
  Packet Statistics: <1/1> <1/1>

- - - The PCIe System has been destroyed - - -

```

**Figure 4.7:** The summary of the output.txt file generated by a single simulated MRd request from EP 2:0:0



- Answering a MWr by writing data to the system memory located locally with the root complex. No packets are returned as Ack-Nak is done in the data link layer:

```
----- SIMULATION COMPLETE -----
                                  @10 us
----- SUMMARY -----

Program was executed with the following parameters:
  ./PCIESystem.out 1 0 0 MWr 0 MWr 10000 -show_full_and_memory 1 -out.txt

Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>

-----
--RootComplex Destructor--
Packet Statistics: <0/0> <1/0>
Requests received time stamps:
    6 ns
--Switch: 000 Destructor--
Packet Statistics: <1/1> <0/0>
--Socket Destructor--
Packet Statistics: <0/1>
--RC Memory Destructor--
- - -Printing RC PCIe Subsystem_Memory- - -
    1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

--EndPoint: 200 Destructor--
Packet Statistics: <1/0> <0/0>
Requests sent with traffic class 5 time stamps:
    0 s
--Socket Destructor--
Packet Statistics: <1/0>

--Switch: 100 Destructor--
Packet Statistics: <1/1> <0/0>

- - - The PCIe System has been destroyed - - -
```

**Figure 4.8:** The summary of the output.txt file generated by a single simulated MWr request from EP 2:0:0

- Answering a large MRd with multiple CplID due to the RCB. 192 bytes is requested from address 48. Note that this results in a transaction that consists of 4 CplIDs just like the RCB example given in subsection 2.1.2:

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
./PCIESystem.out 1 0 0 MRd 0 MRd 10000 -show_full -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <1/4>
Requests received time stamps:
6 ns
Completions sent time stamps:
454 ns 519 ns 584 ns 633 ns
--Switch: 000 Destructor--
Packet Statistics: <1/1> <4/4>
--Socket Destructor--
Packet Statistics: <4/1>

--EndPoint: 200 Destructor--
Packet Statistics: <1/4> <0/0>
Requests sent with traffic class 5 time stamps:
0 s
Completions received time stamps:
454 ns 519 ns 584 ns 633 ns
Average MRd-CplD time is:
633000 ps
--Socket Destructor--
Packet Statistics: <1/4>

--Switch: 100 Destructor--
Packet Statistics: <1/1> <4/4>

- - - The PCIe System has been destroyed - - -

```

**Figure 4.9:** The summary of a large MRd request, proving the RCB functionality

- Answering multiple Requests per simulation:
  - Answering 2 TLPs coming from EP 4:0:0 and EP 3:0:0 where the latter EP is behind an additional packet switch. The same traffic class is used to illustrate the imperative delta-cycle consumption in the switch module.

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
./PCIEsystem.out 0 1 1 MRd 0 MWr 10000 -show_full -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <2/2>
Requests received time stamps:
  2 ns    10 ns
Completions sent time stamps:
 452 ns  694 ns
--Switch: 000 Destructor--
Packet Statistics: <2/2> <2/2>
--Socket Destructor--
Packet Statistics: <2/2>
-----
--EndPoint: 300 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 6 time stamps:
  0 s
Completions received time stamps:
 694 ns
Average MRd-CplD time is:
 694000 ps
--Socket Destructor--
Packet Statistics: <1/1>
-----
--EndPoint: 400 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 6 time stamps:
  0 s
Completions received time stamps:
 452 ns
Average MRd-CplD time is:
 452000 ps
--Socket Destructor--
Packet Statistics: <1/1>
-----
--Switch: 100 Destructor--
Packet Statistics: <1/1> <1/1>
-----
- - - The PCIe System has been destroyed - - -

```

**Figure 4.10:** Simulation log file that illustrates the imperative delta delay consumed by switch modules

- Answering a total of 3 MRd sent simultaneously from the 3 EPs. All endpoints were configured to send packets with different traffic classes. This tests the packet priority arbitration of the system system.

```

----- SIMULATION COMPLETE -----
-----
                                @10 us
-----
----- SUMMARY -----
-----
Program was executed with the following parameters:
./PCIESystem.out 1 1 1 MRd 0 MRd 10000 -show_full 1 -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <3/3>
Requests received time stamps:
  4 ns   10 ns  11 ns
Completions sent time stamps:
  317 ns 701 ns 1157 ns
--Switch: 000 Destructor--
Packet Statistics: <3/3> <3/3>
--Socket Destructor--
Packet Statistics: <3/3>
-----
--EndPoint: 200 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 5 time stamps:
  0 s
Completions received time stamps:
  1157 ns
Average MRd-CplD time is:
  1.157e+06 ps
--Socket Destructor--
Packet Statistics: <1/1>
-----
--EndPoint: 300 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 6 time stamps:
  0 s
Completions received time stamps:
  701 ns
Average MRd-CplD time is:
  701000 ps
--Socket Destructor--
Packet Statistics: <1/1>
-----
--EndPoint: 400 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 4 time stamps:
  0 s
Completions received time stamps:
  317 ns
Average MRd-CplD time is:
  317000 ps
--Socket Destructor--
Packet Statistics: <1/1>
-----
--Switch: 100 Destructor--
Packet Statistics: <2/2> <2/2>
-----
- - - The PCIe System has been destroyed - - -

```

**Figure 4.11:** Simulation log file that illustrates the packet priority of the system, EP 4:0:0 is handled first because of the delta delay in the switch

- Answering multiple b2b requests from one endpoint without relaxing ordering of packets. Clumping is experienced in the RC, as packets experience different delays and are locked in a send order.

```

----- SIMULATION COMPLETE -----
-----
                                @10 us
-----
----- SUMMARY -----
Program was executed with the following parameters:
./PCIESystem.out 10 0 0 MRd 0 MRd 10000 -show_full 1 -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <10/10>
Requests received time stamps:
  6 ns   16 ns   26 ns   36 ns   46 ns
 56 ns   66 ns   76 ns   86 ns   96 ns
Completions sent time stamps:
 448 ns  802 ns  1124 ns 1486 ns 2048 ns
2402 ns 2756 ns 3214 ns 3544 ns 3754 ns
--Switch: 000 Destructor--
Packet Statistics: <10/10> <10/10>
--Socket Destructor--
Packet Statistics: <10/10>
-----
--EndPoint: 200 Destructor--
Packet Statistics: <10/10> <0/0>
Requests sent with traffic class 5 time stamps:
 0 s    10 ns   20 ns   30 ns   40 ns
50 ns   60 ns   70 ns   80 ns   90 ns
Completions received time stamps:
 448 ns  802 ns  1124 ns 1486 ns 2048 ns
2402 ns 2756 ns 3214 ns 3544 ns 3754 ns
Average MRd-CplD time is:
 2.1128e+06 ps
-----
--Socket Destructor--
Packet Statistics: <10/10>
-----
--Switch: 100 Destructor--
Packet Statistics: <10/10> <10/10>
-----
- - - The PCIe System has been destroyed - - -

```

**Figure 4.12:** Simulation log file from a simulation with multiple MRd requests sent with low periodicity, illustrates clumping in the RC

- Answering multiple b2b requests from one endpoint with relaxing ordering of packets. Clumping is not experienced as packets are allowed to enter a wait-storage.

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
./PCleSystem.out 10 0 0 MRd 0 MRd 10000 -show_full 1 -out.txt -plot
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
  Packet Statistics: <0/0> <10/10>
    Requests received time stamps:
      6 ns   16 ns   26 ns   36 ns   46 ns
      56 ns   66 ns   76 ns   86 ns   96 ns
    Completions sent time stamps:
      239 ns 269 ns 301 ns 331 ns 375 ns
      377 ns 393 ns 395 ns 489 ns 495 ns
  --Switch: 000 Destructor--
    Packet Statistics: <10/10> <10/10>
  --Socket Destructor--
    Packet Statistics: <10/10>
-----
--EndPoint: 200 Destructor--
  Packet Statistics: <10/10> <0/0>
    Requests sent with traffic class 5 time stamps:
      0 s    10 ns   20 ns   30 ns   40 ns
      50 ns   60 ns   70 ns   80 ns   90 ns
    Completions received time stamps:
      239 ns 269 ns 301 ns 331 ns 375 ns
      377 ns 393 ns 395 ns 489 ns 495 ns
    Average MRd-CplD time is:
      321400 ps
  --Socket Destructor--
    Packet Statistics: <10/10>
-----
--Switch: 100 Destructor--
  Packet Statistics: <10/10> <10/10>
-----
- - - The PCIe System has been destroyed - - -

```

**Figure 4.13:** Simulation log file from a simulation with multiple MRd requests sent rapidly, multitasking in the RC resolves the clumping

### The Simulation Results for the RC Acting as a Requester

- 1 MRd TLP was sent with a memory address that resides within EP 2:0:0. Correctness of the basic system memory read functionality was ensured by checking that the root complex actually receives a packet containing the data that it requested from system memory. All fields in the packet-header were checked for correctness. The delay modeled in the EP is just set to 800 NS between each CplD returned, in order to check the system functionality.

```

----- SIMULATION COMPLETE -----
                                     @10 us
-----
----- SUMMARY -----
Program was executed with the following parameters:
./PCIESystem.out 0 0 0 MRd 1 MRd 10000 -show_full -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent time stamps:
  1 ns
Completions received time stamps:
  815 ns
Average MRd-CplD time is:
  814000 ps
Data received is:
  1 2 3 4 5
  6 7 8 9 10
--Switch: 000 Destructor--
Packet Statistics: <1/1> <1/1>
--Socket Destructor--
Packet Statistics: <1/1>

--EndPoint: 200 Destructor--
Packet Statistics: <0/0> <1/1>
Requests received time stamps:
  1 ns
Completions sent time stamps:
  801 ns
--Socket Destructor--
Packet Statistics: <1/1>

--Switch: 100 Destructor--
Packet Statistics: <1/1> <1/1>

- - - The PCIe System has been destroyed - - -

```

**Figure 4.14:** Simulation log of the Root Complex using address routing to route a single MRd package downstream to its receiver

- 1 MWr TLP, correctness of the basic system memory write functionality was ensured through observation of the subsystem memory-location written to after simulation termination. The packet had to make its way all the way through the PCIe system fabric based on the location of the system memory address.

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
./PCleSystem.out 0 0 0 MWr 1 MWr 10000 -show_full_and_memory 1 -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/0> <0/0>
Requests sent with traffic class 1 time stamps:
1 ns
--Switch: 000 Destructor--
Packet Statistics: <0/0> <1/1>
--Socket Destructor--
Packet Statistics: <1/0>

--EndPoint: 200 Destructor--
Packet Statistics: <0/0> <1/0>
Requests received time stamps:
1 ns
--Printing PCIe Endpoint Subsystem Memory--
1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--Socket Destructor--
Packet Statistics: <0/1>

--Switch: 100 Destructor--
Packet Statistics: <0/0> <1/1>

-- The PCIe System has been destroyed --

```

**Figure 4.15:** Simulation log of the Root Complex using address routing to route a single MWr package downstream to its reciever



- 1 MRd, a non posted request from the RC is forced to not be handled in the end-point module by hard-programming. This results in a request time out, and the outbound buffer's timer located within the PCIe socket module in the RC replays the request.

```

----- SIMULATION COMPLETE -----
-----
----- @10 us -----
-----
----- SUMMARY -----
-----
Program was executed with the following parameters:
./PCIESystem.out 0 0 0 MRd 1 MRd 10000 -show_full -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class 1 time stamps:
1 ns
Completions received time stamps:
6815 ns
Average MRd-CplD time is:
6.814e+06 ps
--Switch: 000 Destructor--
Packet Statistics: <1/1> <2/2>
--Socket Destructor--
Packet Statistics: <2/1>
-----
--EndPoint: 200 Destructor--
Packet Statistics: <0/0> <2/1>

Requests received time stamps:
1 ns
6001 ns
Completions sent time stamps:
6801 ns
--Socket Destructor--
Packet Statistics: <1/2>
-----
--Switch: 100 Destructor--
Packet Statistics: <1/1> <2/2>
-----
- - - The PCIe System has been destroyed - - -

```

**Figure 4.16:** Simulation log of the Root Complex using address routing to route a single MRd package downstream to its receiver

- 1 MWr, a large write request. The packet is split into multiple packets by the transaction layer within the socket module to satisfy the `max_payload_size` attribute for the PCIe bus. A write request for 256 bytes, which is 1 byte over the `max_payload_size` results in 2 request TLPs.

```

----- SIMULATION COMPLETE -----
                                     @10 us
----- SUMMARY -----
Program was executed with the following parameters:
./PCleSystem.out 0 0 0 MWr 1 MWr 10000 -show_full_and_memory 256 -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/0> <0/0>
Requests sent with traffic class 1 time stamps:
1 ns
--Switch: 000 Destructor--
Packet Statistics: <0/0> <2/2>
--Socket Destructor--
Packet Statistics: <2/0>

--EndPoint: 200 Destructor--
Packet Statistics: <0/0> <2/0>
Requests received time stamps:
1 ns 401ns
--Socket Destructor--
Packet Statistics: <0/2>

--Switch: 100 Destructor--
Packet Statistics: <0/0> <2/2>

- - - The PCIe System has been destroyed - - -

```

**Figure 4.17:** Simulation log of the Root Complex sending a single MWr request for a large amount of data, being limited by the max payload attribute

## The Simulation Results for the RC Acting as Both a Completer and a Requester During the Same Simulation

- 1 MRd TLP was sent from each device in the system. Upstream and downstream Traffic meets on the middle. EPs sends to the RC and the RC sends to EP 002 as a result of system memory address routing.

```

----- SIMULATION COMPLETE -----
----- @10 us -----
----- SUMMARY -----
Program was executed with the following parameters:
./PCIESystem.out 1 1 1 MRd 1 MRd 10000 -show_full -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/1> <3/3>
Requests sent with traffic class: 1 Time stamps:
1 ns
Completions received time stamps:
815 ns
Requests received time stamps:
1 ns 10 ns 11 ns
Completions sent time stamps:
395 ns 629 ns 1079 ns
--Switch: 000 Destructor--
Packet Statistics: <4/4> <4/4>
--Socket Destructor--
Packet Statistics: <4/4>
--EndPoint: 200 Destructor--
Packet Statistics: <1/1> <1/1>
Requests sent with traffic class: 5 Time stamps:
0 s
Completions received time stamps:
1079 ns
Requests received time stamps:
1 ns
Completions sent time stamps:
801 ns
--Socket Destructor--
Packet Statistics: <2/2>
--EndPoint: 300 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class: 6 Time stamps:
0 s
Completions received time stamps:
629 ns
--Socket Destructor--
Packet Statistics: <1/1>
--EndPoint: 400 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent with traffic class: 7 Time stamps:
0 s
Completions received time stamps:
395 ns
--Socket Destructor--
Packet Statistics: <1/1>
--Switch: 100 Destructor--
Packet Statistics: <3/3> <3/3>
- - - The PCIe System has been destroyed - - -

```

**Figure 4.18:** Simulation log of a simulation where all devices are requesters and potentially completers

- 1 MWr was sent from every device. Upstream and downstream traffic meets in the middle. EPs sends to RC and the RC sends to EP 002 as a result of system memory address routing.

```

----- SIMULATION COMPLETE -----
----- @10 us -----
-----
Program was executed with the following parameters:
./PCIESystem.out 1 1 1 MWr 1 MWr 10000 -show_full_and_memory -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <1/0> <3/0>
Requests sent with traffic class: 1 Time stamps:
1 ns
Could not print avg MRd-CplD response delay, number of packets are different
Requests received time stamps:
1 ns 10 ns 11 ns
--Switch: 000 Destructor--
Packet Statistics: <3/3> <1/1>
--Socket Destructor--
Packet Statistics: <1/3>
--RC Memory Destructor--
-- -Printing RC PCIe Subsystem_Memory- - -
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--EndPoint: 200 Destructor--
Packet Statistics: <1/0> <1/0>
Requests sent with traffic class: 5 Time stamps:
0 s
Requests received time stamps:
1 ns
-- -Printing PCIe Endpoint Subsystem Memory- - -
0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--Socket Destructor--
Packet Statistics: <1/1>
--EndPoint: 300 Destructor--
Packet Statistics: <1/0> <0/0>
Requests sent with traffic class: 6 Time stamps:
0 s
--Socket Destructor--
Packet Statistics: <1/0>
--EndPoint: 400 Destructor--
Packet Statistics: <1/0> <0/0>
Requests sent with traffic class: 7 Time stamps:
0 s
--Socket Destructor--
Packet Statistics: <1/0>
--Switch: 100 Destructor--
Packet Statistics: <2/2> <1/1>
-- - - The PCIe System has been destroyed - - -

```

**Figure 4.19:** Simulation log of a simulation where all devices requests memory writes to the other devices

### The Simulation Results for the Test Where Computation of the n'th digit of $\pi$ is Outsourced to the EP

Figure 4.20 shows a simulation summary where the n'th digit of pi is calculated using Bellard's algorithm and sent to the RC every delta cycle. Computational power is simulated to be outsourced.

```

----- SIMULATION COMPLETE -----
@1 ms
-----
SUMMARY
-----
Program was executed with the following parameters:
./PCIESystem.out 900 900 0 MWr 0 MRd 1000000 -show_full_and_memory -out.txt
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <900/0>
First packet received @ 4 ns

Last packet received @ 359604 ns
--Switch: 000 Destructor--
Packet Statistics: <900/900> <0/0>

--Socket Destructor--
Packet Statistics: <0/900>

--EndPoint: 200 Destructor--
Packet Statistics: <900/0> <0/0>
First packet received @ 4 ns

Last packet sent @ 359600 ns
--Socket Destructor--
Packet Statistics: <900/0>

--RC Memory Destructor--
-- -Printing RC PCIe Subsystem_Memory-- - -
(1) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(2) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 4 1 5 9 2 6 5 3 5 8 9

(30)5 2 2 3 0 8 2 5 3 3 4 4 6 8 5 0 3 5 2 6 1 9 3 1 1 8 8 1 7 1
0 1 0 0 0 3 1 3 7 8 3 8 7 5 2 8 8 6 5 8 7 5 3 3 2 0 8 3 8 1
4 2 0 6 1 7 1 7 7 6 6 9 1 4 7 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--Switch: 100 Destructor--
Packet Statistics: <900/900> <0/0>
- - - The PCIe System has been destroyed - - -

```

**Figure 4.20:** Simulation log of a realistic PCIe system scenario

During all these tests, the system was also indirectly tested for correctness on displaying packet traffic statistics.

## 4.2.2 Delay Model Accuracy

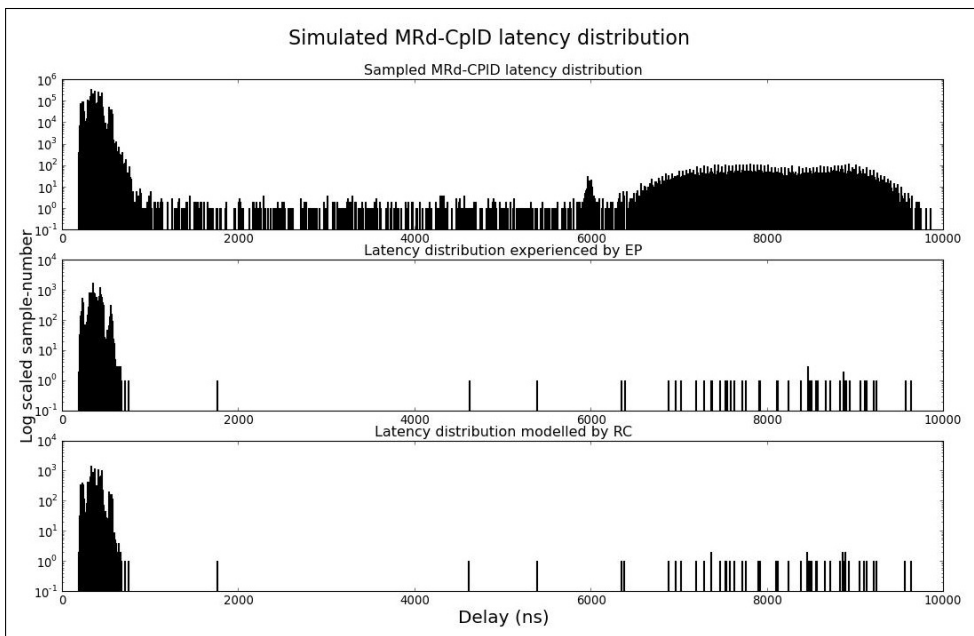
- The plots in figure 4.22 shows the sampled latency distribution from figure 4.2 versus the generated latency distribution from the simulation from log 4.21. The graphs provide a good illustration of the MRd-CplD delay accuracy of the RC. The middle graph in figure shows the samples that are randomly drawn from the delay file in the Root Complex module upon reception of MRd requests from the endpoint. The graph at the bottom represents the delays that are experienced for a simulated endpoint module. The experienced delays for the endpoints are calculated using time stamps with key-vectors in t. TLP simulation tags in the TLP class are used to correlate the incoming CplDs with the requests in the software layer as the outbound buffer is located in the transaction layer.

```

----- SIMULATION COMPLETE -----
----- @1 ms -----
----- SUMMARY -----
Program was executed with the following parameters:
./PCIESystem.out 0 15000 0 MRd 0 MRd 1000000 -show_full 1 -out.txt -plot
-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>
-----
--RootComplex Destructor--
Packet Statistics: <0/0> <15000/15000>
--Switch: 000 Destructor--
Packet Statistics: <15000/15000> <15000/15000>
--Socket Destructor--
Packet Statistics: <15000/15000>
--EndPoint: 300 Destructor--
Packet Statistics: <15000/15000> <0/0>
Average MRd-CplD time is:
394492 ps
--Socket Destructor--
Packet Statistics: <15000/15000>
--Switch: 100 Destructor--
Packet Statistics: <15000/15000> <15000/15000>
- - - The PCIe System has been destroyed - - -

```

**Figure 4.21:** Simulation log of 15000 packets over 1 ms

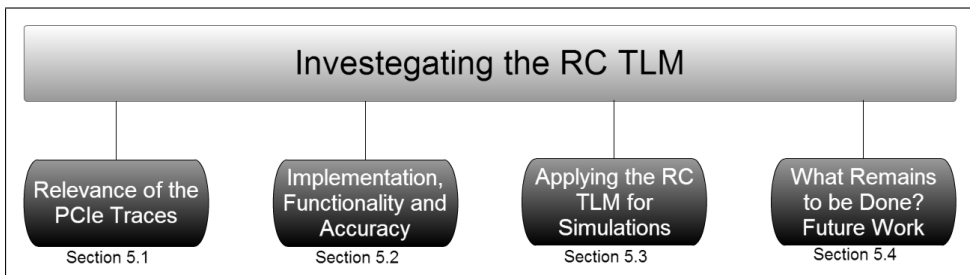


**Figure 4.22:** Correlation between sampled and simulated latency distributions





# Discussion



**Figure 5.1:** Topics of discussion and their sections in this chapter

This chapter discusses the obtained results from the PCIe traffic traces, as well as their relevance to the general RC TLM. The following section also investigates the decisions that were taken and the simplifications that were made during the implementation of the RC TLM, their impacts on the statistical realisticness of the model are evaluated. A description is given in section 5.3 of how to utilize the RC TLM as a system environment to do performance simulations of PCIe EPs. Finally, the work that remains to be done on the RC model is itemized.

## 5.1 Analyzing the PCIe Trace Recordings

The application that was executing on Oracle's in-house made EP during the PCIe traffic tracing had a heavy read and heavy write nature. A total of 5 GB worth of .pex files were gathered during the recording. These were analyzed and converted into around 14MB of MRd-CplD delta delays. Such a large data sample satisfies the law of many and is therefor a realistic representation of the delay data specifcly for the SUN PCIe RC running the Qperf DMA test. The EP

and the RC utilized traffic class 0 only, with snooping and with strict ordering. A constant traffic scenario restricts the area in which the RC TLM can be utilized for performance simulations. Further recording of PCIe traffic is needed to complete the TLM of the SUN FIRE X4170 PCIe RC, to make it more flexible for different load scenarios. Variations in the traffic class, the snooping attribute, the relaxed ordering attribute and for newer PCIe architectures; TLP hints and extended tags, are expected to deeply influence the performance of a PCIe system. The Latency distribution of MRd-CpID traffic from the samples as well as limitations of extraction approach are discussed below.

### 5.1.1 The Recorded MRd-CpID Latency Distribution

The latency distribution that was extracted from PETrace exports is shown in figure 4.2 in section 4.1.2. A logarithmic y-axis in the histogram reveals that the latency distribution consists of two major sub-distributions. The packets that arrives between the minimal arrival time of 184 ns, and 1000 ns create the heavy tailed distributed as expected from routing theory. A quick run of a Matlab function called ALLDISTFIT with the 3659394 MRd-CpID deltas as argument, reveals that the Nakagami distribution is the parametric probability distribution that best fits the dataset, if the set were to be modeled with a statistical model.

The 13696 packets that arrive between 6000ns and 10000ns make up a small normally distributed sub-section. Running Matlab's ALLDISTFIT, reveals that the generalized extreme value distribution distribution is the parametric probability distribution that best fits the sub data set.

The MRd-CpID samples include payloads of multiple sizes, the payload fluctuations are delay-modeled indirectly through the delta delay file which is adequate for a TLM. Same goes for:

- Flow control halts.
- Cache snooping halts.
- Strict ordering halts.
- Violations to the read completion boundary.
- Violations to the maximum payload size.

They all introduce jitter in the transactions that results in the latency distribution iin the histogram in figure 4.2. So by including the sampled delay file, all these are indirectly modelled from a transaction latency perspective.

### 5.1.2 The Delta Delay Extraction tool

The C++ script that was written to generate delta delay files from PETrace output is attached in appendix B. It converts already exported text files from the

PETrace software to a format that contains 1 delta delay per column using a simple terminal execution. Its output files are easily inserted into the RC TLM by adding it to a relative folder and by reading it into the model during the construction phase of the PCIe system. The software is specialized on extracting simple MRd-CPLID delta delays. Handling of more complex traffic scenarios, i.e when multiple completions are returned, is not yet implemented in this extraction software.

## 5.2 Analyzing the RC TLM

This section reasons for the decisions that were made during the implementation of the PCIe TLM. The test results are also analyzed to ensure correctness of the RC TLM with respect to functionality and to timing accuracy.

### 5.2.1 Implementing the RC TLM

The selection of a realistic delay is discussed in this subsection, so is the choosing of SystemC as a programming language, reasoning behind implementation of the RC adjacent memory as a dedicated module and finally simplifications that were made to create a PCIe RC model with a high abstraction level.

#### Choosing the delay modeling approach

A rational request-to-completion delay model is imperative considering the hardware-realistic accuracy of the RC model. The delay-correctness for MRd-CplID for the RC model was ensured by using real hardware traffic traces from the Teledyne Lecroy tracer. Two implementation variants of the RC delay-module were considered, both utilizing the wait statement in SystemC. The models that were considered were:

- A trace based delay model that is statistically correct because it deploys the actual data from hardware traces for delay modeling. Request packets are answered with completion packets after a randomly drawn sample delay from a database of real hardware traces.
- A mathematical delay model that provides delay-samples drawn from an estimated delay distribution, typically a long-tailed distribution. A superposition of a long-tailed distribution together with a normal distribution was also an option to include a model for the sub distribution shown in the spike between 6000ns and 10000ns in the sampled latency distribution shown in figure 4.2. The statistical model was to be correlated and calibrated with real hardware through PCIe traffic traces. Request packets

are answered with completion packets after a delay given by a statistical model of the completion delay.

The first of the two was chosen as the delay model method mainly for its accuracy, but also because of its simple structure compared to the statistical complexity that would have arisen for the latter approach. Statistical modeling of the delay samples might come in useful if an expansion of the model with delay data for traffic classes and such turns out to create too large data storage demands.

The observations of sub-distributions discussed in section 5.1.1 supports the approach given in the second bullet above, of delay modeling using probability distributions instead of trace samples directly. Matlab provides alongside with the best fitting probability distributions, parameters for correctly modeling the data that it takes in using the listed probability distributions. These parameters can be inserted in the Nakagami and the GEV probability density functions. Finally the two delay models can be fused by weighting each probability and adding them together using the formula from section 2.2.2.

$$f_{total}(\dots) = p_{Nakagami}(\dots) w_{Nakagami} + p_{GEV}(\dots) w_{GEV}$$

**Figure 5.2:** Mixture probability distribution function of the MRd-CplD delays

Figure 5.2 shows a the alternative to the direct sample delay modeling. Advantages with a TLM using statistical models instead of brute forced data samples are mostly related to less memory consumption. The delays in the RC are calculated upon reception for each completion instead of being read from constructor loaded vectors of samples, in this way both simulation initialization time as well as execution time consumption could be saved by implementing this statistical model. These benefits are not significant enough when comparing it to the advantages of the brute-force approach with a brute-forced nature of details. Nanosecond accuracy is achieved by fetching MRd-CplD latencies from the sampled delta delay files, this mimics hardware ideally with the precision of the hardware tracer serving as the only bottleneck for timing accuracy.

### Choosing SystemC as the modelling language

The objective of the model was to be suitable for performance measurements in the transaction layer domain. Details of PCIe protocol in lower abstraction levels was to be ignored in order to create a quick functional RC model that is accurate with respect to delay modeling and general functionality. The modeling language that was to be selected had to provide a simulation engine with an efficient and robust notion of time. The language also had to be flexible in terms of connecting the resulting RC model to already existing testing software.

Object orientation was desired since inheritance was found useful for dealing with various types of packet-headers that derives from a generic unit.

Languages like VHDL and Verilog were considered to be way too low level oriented, and were found to have poor performance compared to higher level languages such as java and pure C++ for simulation. High level languages do not provide simulation kernels and thus have no notion of time when it comes to simulation. SystemC on the other hand is as described in section 2.4 ideal for modeling hardware using TLM. Multiple arguments were found for using SystemC to model the RC, some of these are listed here:

- SystemC is well structured with the module macro. Internal module processes are encapsulated within their respective modules and creating sub-modules and reusing modules is a trivial task.
- The degree of C++ in the SystemC model can be chosen after what is beneficiary since SystemC is a superset of C++.
- The SystemC model can easily be interpreted by software coders at Oracle because of its resemblance to pure C++ and other non-modelling languages.
- SystemC as a c++ framework provides a high level of abstraction, its object orientation is useful for dealing with inheritance from generic TLP-headers.
- True random extraction of delay data is easily done with C++'s srand and text file manipulation.
- SystemC is publicly available, there is no need for licenses or royalties. It is both low cost, and easy to use. A normal C++ compiler can be used to generate executables.
- SystemC is accepted as a modeling language in the industry.
- Memory leaks and thus run-ability can be checked using the Valgrind mem-check tool

### **Separating RC's Memory in a Dedicated Module**

A model for the RC subsystem memory was implemented as a separate module to uncover memory-related causes of performance degradations, such as RCB, max payload size and so on. By having a dedicated module representing the memory instead of including the memory as an array within the RC itself, one can model the delay of the actual memory transfer per byte in a more fine

grained manner. For larger reads it is more convenient to model the RC subsystem memory as an array directly within the RC model itself because of access cycles. Both approaches are implemented and toggling between the two memory modeling systems is easily done with boolean program settings.

### **Simplifications**

The internal complexity of the RC model has been simplified in order to reach the deadline of this assignment:

- The memory-model connected to the RC is a simple array containing integers that represent bytes, addressing the module is done in a byte aligned manner. The degree of detail in the transaction level RC model has been a repeated question throughout this thesis. Intel's Quick Path Interconnect was originally intended to be modeled in detail within the RC model, the QPI protocol is used between memory, CPU and the RC itself. With such an approach, the RC would have been a bridge from one packet based protocol to another, allowing fine grained performance modeling of the entire system resulting in a more flexible tool for performance simulations. However, since the only available protocol analyzer was the Lecroy tracer, this system complexity was abstracted away from the model whose only task is to provide hardware realistic feedback to the EP.
- System configuration was made abstract in the RC model. The system is configured in an outer SystemC module, configuration registers are set through module constructors, removing the need for configuration requests. All devices are constructed with device ID's and BARs. Performance testing is done on an already configured system. Hot-plugging of PCIe devices is also a functionality that is not needed for modelling of an RC with respect to transaction performance.
- TLPs are the only type of packets that are supported, DLLPs are ignored as the goal of this thesis is modeling of an RC in the transaction domain. This means that posted requests are not delay-modeled because the chain is broken within the RC, and no completion packets are returned. Flow control is also not modeled in this thesis, VCB are implemented with vectors, and have unlimited restrictions. However since flow control is such an important metric to surveill for monitoring system performance, it can be easily implemented using help variables or directly readable VCBs in the system.
- The RC's TLP compatibility implemented so far is restricted to memory read requests, completions and memory write requests. The data types of these packets are structs that are derived from the generic TLP class.

Other TLP types with different values indicated within the header type-field, that are sent to the RC, will be ignored in the RC's handling thread. Support for new TLP types can easily be added by creating new children of the generic header. Handling of these can also easily be added in the switch-case statement within the RC.

- The size of the RC's system memory fraction is reduced to lower program memory-consumption. If one were to allocate an array that is the size of 64 bit of addressable memory without using stored files representing memory, one would need a large amount of random access memory on the simulation hardware. This simplification however should not affect the RC's ability to perform as a PCIe environment for performance reasons as the data that is stored in the simulated memory is of little importance. One can loop through the small amount of memory several times instead, in a barrel roll fashion.

### 5.2.2 Testing The Functionality

The results from testing the PCIe-system described in section 3.17 with 3 EPs, a switch instance and the RC module gave the results described in section 4.2. The implications from these results are described in this section, demonstrating the functionality correctness of the RC TLM.

#### The RC as a Completer

The first few tests that were run, tested the basic functionalities of the RC as a completer.

**RC Answering Single Requests** Figure 4.7 shows a simulation summary of a simple memory read request that was sent from EP 2:0:0 to the RC and its corresponding completion packet. All the counters that are implemented within the modules were printed out in each module's destructor, describing the amount of packets sent received or forwarded by the respective modules. The send/receive/forward statistics was used together with verification of the data that was received in the EP to conclude that the memory read request functionality is adequate from an EPs point of view. The packet sent and the packet received time-stamp from the EP time-stamp vectors was used to calculate the average MRd-CplD time consumption. For this specific simulation consisting of only 1 MRd-CplD of 0 and 328 ns the average read to completion latency was 328 ns, which is within reasonable boundaries of the total sample average MRd-CplD of 393 ns as shown in table 4.2. In short, the memory read request functionality is correctly answered by the RC TLM.

Figure 4.8 lists the simulation summary of a simple memory write request sent from EP 2:0:0 to the RC, this summary proves the general correctness of the RC as a completer of memory write requests. A small fraction of the RC adjacent memory was printed to observe that the changes made to the RC memory during the simulation were correct. The 10 bytes that were written to the RC memory using 1 MWr request corresponds to the requested data payload, the entire RC memory was originally constructed to be all 0s. Detailed information of the write request data output log is not showed in this summary, however studies of the entire program output reveals this fact. In short, the correct actions are taken by the RC TLM from a memory write request.

Figure 4.9 lists the simulation summary of a memory read request for a large amount of memory sent from EP 2:0:0 to the RC, this summary proves the general correctness of the RC as a completer of large memory read requests. 192 bytes of data were requested from address 48 of the RC memory, the data received in the EP is not shown in the summary for formatting reasons. It can be observed that EP 2:0:0 sends 1 request, and yet receives 4 completion packets. This is due to the read completion boundary at 64 bytes. The CplD was split up in the software layer of the RC, first packet from address 48, turned out to contain 16 bytes, the two next ones were 64 bytes each, and the last one was 48 bytes, resulting in a total of 192 bytes with packets aligned at the 64 byte RCB. The functionality of the RCB completion-splitting proves to be correct as the result matches the example given in the theory section 2.1.2, the timing on the other hand is not yet implemented in a hardware realistic fashion as further packet tracing is needed. At this point the first packet is returned after a correctly drawn random sample, however the distance between the successive packets in nano seconds, is not yet modeled. Neither is completion combining during VCB stalls.

**RC Answering Multiple Requests** The simulation summary in figure 4.10 lists the result of EP 3:0:0 and 4:0:0 both requesting a memory read simultaneously. The switch in between EP 3:0:0 and the RC consumes a few additional delta cycles for routing, because of this extra travel latency, EP 4:0:0 is guaranteed to be served first by the RC no matter what its TC is. This is when strict ordering is used, and no multitasking in the RC is enabled.

Figure 4.11 shows the simulation summary of the simulation where all three EPs were simultaneously issuing a MRd each at 0ns. EP 2:0:0 use TC 5, EP 3:0:0 use TC 6 and EP 4:0:0 which is connected directly to the RC use TC 4. The interesting fact to note about this simulation result is that packet priority is correctly demonstrated within the first switch. The MRd packet from EP 2:0:0 and from EP 3:0:0 arrives simultaneously at switch 1:0:0 at 0 ns, the packet from EP 3:0:0 is forwarded first while the packet from EP 2:0:0 is stored in virtual channel buffer



5. This can be observed from the chronological order of the completion arrival times as strict ordering and no multitasking is used in the RC module. Even though EP 4:0:0 utilize the lowest traffic class, the delta cycles used for handling in the switch between EP 2:0:0 and EP 3:0:0 causes its request to be handled and returned first. Priority inversion is experienced in the RC application layer as neither preemption nor multitasking is enabled in the RC. The MRd-CplD latencies were stacking for the requests that were stuck in virtual channel buffers while the RC was handling the packet from EP 4:0:0 and then EP 3:0:0. EP 2:0:0 experienced a MRd-CplD latency about 3 times greater than of EP 4:0:0.

The next two test results further investigates the phenomenon of RC request ordering and clumping of requests experienced in the previous paragraph. Figure 4.12 in the result chapter lists the summary of a simulation where EP 2:0:0 sent multiple MRd requests back to back, only 10 ns passed before successive requests were issued. With a minimum sampled MRd-CplD of 184 ns, all 10 requests were guaranteed to be issued before a single completion was returned. It is important to note the high accumulated completion times for the last requests. The first request is answered after 448 ns, the second after 790, and the last request is answered after 3754 ns, 3664 ns after its issued request. No multitasking in the RC and request clumping, causes the average MRd-CplD time to become 2.11  $\mu$ s.

To allow for a more dynamic RC model, multitasking was implemented as a TLM setting within the RC, figure 4.13 shows the simulation summary of a simulation performed with multitasking enabled. The same request scenario is sent from EP 2:0:0 as that of the previous test scenario, 10 requests are sent every 10 ns for the first 90 ns of the simulation time. Each request is received in the same order by the RC, once each request enters the handling queue in the application layer of the RC, data is gathered from the memory and a corresponding completion packet is built in the same cycle as the receive VCB selection. Once the packet has been built, a return time stamp is calculated using the delay sample database, and added to a vector of timestamps to perform packet sending. The completion packet is also added to a vector with a key that maps directly to that of the send time stamp vector. A dedicated thread for answering requests is activated once every delta cycle to check if the current delta cycle has a packet to return, that packet is then added to the send queue. The result in figure 4.13 shows that the average simulated MRd-CplD time represents that of the average sampled MRd-CplD delay, giving the RC model a functional extra-feature for performance simulation of EPs that are rapid requesters.

### **The RC as a Requester**

The RC's basic functionality of initiating requests both reads and writes on behalf of the CPU are shown in figures 4.14 and 4.15. These simulation logs

demonstrate the switches ability to utilize their configuration registers and perform address routing based on BARs towards correct EPs. Memory located in EP 2:0:0 was successfully read in the first summary, this can be seen by comparing the data that was returned with the construction of the EP memory. The MRd-CplD latency is modeled with the arbitrary value of 800 ns, as this thesis involves the construction of an RC TLM and not an EP TLM. The memory was also successfully written to judging from the sent payload compared to the resulting memory after a write to a memory that is constructed to all 0s.

The simulation log of figure 4.16 proves the functionality correctness of the outbound request buffer in the PCIe socket of the RC model. A non-posted memory read request was issued from the RC towards EP 2:0:0. The EP module was reprogrammed to discard the completion of the first request, forcing the request in the outbound to be replayed after an arbitrary timeout period which was set to 6000 ns. The new request was answered right away resulting in an average MRd-CplD delay of 6814ns. Further tracing of target platform's RC traffic is required to fill inn a realistic timeout constant to complete the TLM model of the RC.

Figure 4.17 shows the packet trace summary of a simulation where 256 bytes of data was requested to be written with a single write initiation from the RC application layer. It can be observed from the result that the function of the `max_payload_size` parameter of 255 bytes is activated within in the PCIe protocol stack module, the packets sent counter of the socket destructor is equal to 2 while the packets received from the RC device core is 1. The maximum payload size parameter has a large influence on the performance of PCIe devices, overhead of the entire sending is increased with the size of the header and the ECRC field. Generally the packet should be sent right away after VCB arbitration since the packet splitting is initiated in the transaction layer.

### **The RC as Both a Requester and a Completer**

Finally the RC's functional accuracy was ensured by observing the resulting simulation summaries of traffic scenarios where requests move both upstream and downstream. Firstly all EPs including the RC was simulated as requesters for memory reads, 1 MRd per device, the simulation summary in figure 4.18 shows the resulting traffic load on the system. The switches performed priority arbitration upstream and address routing downstream, which resulted in the arrival of the request from the RC at EP 2:0:0 first of all as the downstream routing takes 0 ns due to no buffer arbitration. EP: 4:0:0's request arrives first of the EP's requests, the request from EP 3:0:0 second and the request from EP 2:0:0 due to traffic classes and switch delta cycles. Similar is the traffic scenario of the trace summary where all devices are requesters for memory writes, this is shown in figure 4.19. It can be concluded from these two output log summaries that the

RC performs as expected when both answering and issuing requests in the same simulation.

### **A Realistic System Scenario**

The realistic test scenario of the PCIe system consisted of an EP continuously calculates the  $n$ 'th digit of pi using Ballard's algorithm. The data is written to the host memory once it is calculated every cycle. This test was meant for illustrative purposes of the system capabilities. The host CPU is freed from computational loads and can use its resources for other tasks. The resulting output summary is shown in figure 4.20. The result shows that the RC's memory is successfully written two with 900 decimals of PI.

### **Simulated MRd-CplD Latency Accuracy**

The terminal output of the simulation with 15000 MRd-CplDs in figure 4.21 together with the auto generated plot in figure 4.22, proves the correctness of this TLM approach. This result is probably the most important to note throughout this thesis. Both the heavy tailed fraction and the bell shaped distributions are represented in both the RC's drawn samples, as well as in the EP-s experienced MRd-CplD latencies. Slight differences can be observed between the simulated delays in the RC's and the simulated experienced delays of the EP, because of accumulated delta cycles for switch arbitration. These small differences are so insignificant that they are negligible when considering the delay accuracy of the RC TLM. The small delays accumulated within switches for arbitration prove to be irrelevant to achieve realistic latency modeling. All in all, the MRd-CplD Latency accuracy perfectly replicates that of the gen. 1. RC within the traced sun server.

Support for other RC architectures would include new MRd-CplD trace collections, since traced latencies are hardware specific. Each RC architecture performs differently just like PCIe EPs are expected to do during performance simulation. The degree in which the MRd-CplD delays vary for various RCs remains to be unknown, the delta delays recorded on the SUN server might still provide a decent pointer for performance simulations.

## **5.3 Using the RC model for performance testing of EPs**

To unlock design-phase performance testing of PCIe EPs the EP's needs to be modeled in a manner that is compatible to the RC TLM. Typically, this would be in the shape of SystemC modules like the one that was created for functional verification of the RC TLM.

### 5.3.1 Deciding on the PCIe performance criteria

The performance of a PCIe device depends on multiple variables, both within the device itself and from its switch based system-environment. The variables of interest for performance measurements should be fixed in advance of the SystemC implementation of the EP module. The device under test can be checked for its performance as a completer, a receiver, or both a completer and a receiver. The system performance criteria for the simulation can be inbound reads, inbound writes, outbound reads, outbound writes, receive latency, transmit latency or round-trip latency. One should also map whether the performance testing targets testing of the application running on the EP or simply PCIe protocol that is implemented on the EP. Bandwidth simulations should indicate sustainable values, not peak to peak performance. The module can be significantly simplified with respect to the areas of interest of the performance simulation that is to be executed.

### 5.3.2 Implementing a TLM of the EP

The model of the EP should be made as a dedicated SystemC module in the transaction domain. If the performance test targets the EP application, then the socket module within the RC TLM might be re-used to simplify the EP TLM implementation. The SystemC TLM of the PCIe EP should be created to fit the template in figure 5.3.

The code in figure 5.3 is a template for implementing a TLM of the EP under test. The figure shows a stripped version of the EP module that was used to test the RC functionality. Aspects worth noting for the EP implementation and for setting up the PCIe system are given in the following bullets:

- The EP module needs to support one in-port and one out-port, both for carrying TLP objects in a similar manner to the RC, these are displayed in figure 5.3.
- The EP should be designed as a transaction layer model. System workarounds are crucial for functionality that relies on abstraction-layers lower than the transaction layer.
- The EP should mimic the actual functional design of the EP to allow parameter tweaking.

Following the EP TLM template given in this section allows direct connection to the RC TLM using SystemC specific information carriers of the TLP type. Single or multiple instances of the EPs are allowed to be connected to the RC. The EP module and the outer module interconnecting all the PCIe devices should include functionalities and variables for monitoring performance and to differ

```

/**
 * |----- PCIe-EP -----|
 * |----- FUNCTIONALITY -----|
 * |----- v ^ -----|
 * |----- PCIe Socket -----| -> Upstream
 * |----- -|< -----| Downstream
 */SC_MODULE(PcieEndPoint) {
    PcieSocket *          socket;
    sc_in<TLP>            incoming_tlp_to_socket; //in
    sc_out<TLP>           outgoing_tlp_from_socket; //Out
    sc_signal<SoftwareData> to_socket_from_dev_core; //internal
    sc_signal<TLP>        from_socket_to_dev_core; //internal

    //Endpoint specific functions
    void EndPointFunctionality();
    void ReceivePacketAction();

    //EP Memory
    int endpoint_memory[ENDPOINT_MEMORY_SIZE];

    SC_HAS_PROCESS(PcieEndPoint);
    PcieEndPoint(sc_module_name name_,[conf-reg param...]) : sc_module(name_) {
        //Generic PCIe Socket
        socket = new PcieSocket("EP_1",[conf-reg parameters...]);
        //Basic EP Threads
        SC_THREAD(EndPointFunctionality);
        SC_METHOD(ReceivePacketAction);
        sensitive << from_socket_to_dev_core;
    }
    ~PcieEndPoint() {
        //Analyze simulated data
        delete socket;
    }
};

```

**Figure 5.3:** A blue print of a generic EP TLM to ensure RC TLM compatibility

between the actual total payload and the amount of bytes received to calculate the net bit-rate instead of the gross bit rate. Typical examples of such variables are time-stamp vectors for storing the timestamps for all traffic that is simulated, packet counters, payload counters, memory surveillance and so on.

### 5.3.3 Running the Performance Simulations

Simulation is executed using the `sc_start` function after having instantiated all desired modules in a system and interconnected them. Simulation examples are demonstrated in the main file of the test setup. The RC was tested for memory leaks using Valgrind, see appendix F for a screen-shot of the memory check summary. A model that is free from memory leaks enables long lasting performance simulations using the RC module. Measuring memory leaks on the SystemC library was found to require that the SystemC library is compiled with the `p-threads` flag, false positives on memory leaks were originally discovered be-

fore this fix. The definitively lost memory counter was 0 for the system after the p-threads fix, the other parameters experienced false positives due to Valgrinds limited support for SystemC, they should in reality be equal to 0. Simulations of up to the arbitrary value of 15000 MRd-CplD packets over 1 milliseconds of simulation time have been executed successfully using the RC TLM.

### 5.3.4 Evaluation of the RC TLM as a Tool for Performance Testing

A realistic simulation environment for a PCIe EP performance contributes to locating optimal system solutions that lower silicon size, power consumption, and development and verification costs [25]. An example is finding a minimum size of a buffer by maximizing performance for a selected system performance metric. Reducing buffer sizes to the minimum point where system functionality still remains unaffected, reduces the number of redundant buffer cells, thus reducing chip size, which again reduces power consumption. A good simulation tool removes the need for numerous hardware prototypes thus reducing development and verification costs.

Performance simulation of a PCIe EP TLM using the RC TLM developed in this thesis has the potential to reveal causes of performance degradations related to the following PCIe parameters:

- Parameters affecting bandwidth
  - Maximum Payload Size
  - Maximum Read Request Size
  - Completion Combining in the RC
- Parameters affecting packet transfer rate directly
  - Header Credits. Receive buffer size for header.
  - Data Credits, Receive buffer size for data payload .
  - Tag space limitations

The RC model created in this thesis is found to be statistically equal to a hardware version with respect to read transaction round-trip latency. This is very useful for evaluating EP read request size since the size of the outbound buffer size decides the maximum number of plausible outbound memory read requests. The size of the outbound request buffer should be larger for systems that have large MRd-CplD latencies. This is specially useful for the extended tag feature of the third generation PCIe with 256 being the roof of outbound requests instead of 32 for first generation PCIe systems.

The implemented model for the RC does not model completion combining directly, however this is indirectly in-cooperated in trace variations in MRd-CplD latencies as discussed in section 5.1.1. A traced completion that is the

result of completion combining appears to have a slightly longer MRd-CplD latency. The second section of the MRd-CplD distribution is expected to partially consist of latency samples from packets that have experienced RCB splitting and then read completion combining.

Some workarounds should be implemented for having a TLM that excludes the data link layer. Header credits and payload credits could be necessary to monitor. These could be implemented as integers in the module level of the RC and the EP TLMs. Once this is implemented, sending of packets can be restricted by header and data credits. Optimal queue depths of VCBs can also be found.

To sum up, the RC provides an accurate delay module. Help- variables such as time-stamp vectors and packet counters can be used to monitor and later improve the net bit rate, the latency round-time and the jitter of the PCIe EP.

## 5.4 Future Work

The bullets below summarize the discussion with respect to what remains to be done on the model in future work.

- **More details in the functionality aspect.** Engineers at Oracle are now developing third generation ASICs for EPs, header should be modified to include extended tags(ET) and TLP processing hints(TPH) which are new for third generation PCIe architectures. Extensions of the RC TLM should also include support for steering tags(ST), address translation services(ATS). Another functional extension that is useful in virtual database modeling would be single root I/O virtualization (SR-IOV), it allows a single endpoint to be virtualized to appear as multiple separate physical PCIe devices.
- **Further Modularity of the System.** For a more detailed simulation, accumulative delays from various performance degradations should be modeled. The memory structure of host should be modeled in detail. The RC TLM would then typically include a PCIe to QPI bridge. And details about memory cache-lines and dw aligned memory.
- **EP to EP communication.** Currently the switch modules are only able to switch traffic from EPs upstreams, and from the RC downstream. The EPs are not able to communicate with each-other. If this is desired for the performance simulations, then such a functionality can easily be implemented by slightly modifying the switch module.
- **Support for exchange of buffer credits.** The lower abstraction levels contain important features for overall performance. These can be modeled in a high-level manner with for instance; simple integers representing credit flow with DLLPs.
- **Record more traffic.** Round-trip latency distributions vary for different architectures and traffic scenarios. Latencies should be mapped and added to the RC

TLM for all TCs for the expected simulation scenario. The exact traffic load currently used for performance measurements with QEMU should be sampled on a relevant server architecture.

- **Optional Future Work**

- **Prediction models to save memory consumption.** An option for future work is to record large amounts of traffic and create prediction models from traffic patterns that correlate parameters such as traffic class, system traffic load and payload length variations, all for transfer latency. The RC model then calculates the round-delay for insertion by using the prediction model. Generally more research is needed for covering variation in traffic scenarios with the RC TLM.
- **Patch QEMU with the delay model** An option for future work is to patch QEMU with a realistic delay model, based on the very principle described in this thesis to avoid verification of the RC TLM tool implemented in this thesis. The ich9 chip-set is already emulated with detailed functionalities.



## Conclusion

Having a realistic simulation environment for PCIe endpoints(EPs) is favorable as a PCIe design engineer. In this thesis such an environment was created consisting of a transaction level model(TLM) of a PCIe root complex(RC).

Modeling a PCIe device in detail is cumbersome due to the immense nature of the layered architecture. The model was simplified to target the transaction layer domain, lower protocol layers were abstracted away from the model.

The implementation of the TLM focused on statistical correctness of transaction latencies. It was implemented with the C++ interface, SystemC, making the model easy to understand and to modify for both software and hardware engineers. The SystemC TLM is able to model delays with high accuracy having the user-defined SystemC simulation time-quantum as a maximum value for timing error.

The communication infrastructure of the TLM was made for basic transactions of transaction layer packets(TLPs). The TLPs can be routed to indicated destinations, prioritized and treated with respect to their header descriptions.

The RC TLM imitates real PCIe RCs by incorporating both the timing aspect and the functionality aspect of a PCIe RC. A realistic delay principle was implemented. It that randomly draws samples from a latency database whenever the TLM simulates a delay between a request and it's corresponding completion(MRd-CplD). The database consist of traffic samples from existing RC hardware. Support for a specific target server platforms may be achieved by recording PCIe traffic on the target architecture and including the resulting data in the TLM.

By having a realistic variation of delay samples instead of a constant delay value for each transaction, system jitter is also modeled. The uniformly distributed probability of the random number function in C++ ensures that completely random samples are drawn which prevents distortion of the sampled latency distribution.

The RC TLM supports interconnection of PCIe TLMs with I/O ports of the TLP data type. Endpoints may connect directly to the RC's multi-ports, acting as requesters or completers, or both during a single simulation.

A LeCroy protocol analyzer was used to sample PCIe traffic between an EP and a SUN FIRE x4170 m2 server with two Xeon processors. The sampled traffic was exported and processed into the format supported by the RC TLM delay model. The delay database was created with approximately 4000000 delay samples, distributed in a heavy-tailed fashion.

The results from the functionality test-runs prove the correctness of the modeled transaction system architecture. The RC TLM responds correctly to requests as a completer. It is also able to issue requests as a requester and combine received completions with outbound requests. The results demonstrate the correctness of all sub-modules in the root complex TLM, including the PCIe protocol stack and the switch module.

The basic transaction functionality allows packets to be routed to and from the RC within few delta cycles which is negligible to the simulated system latency. The results from test-runs focusing on timing, prove that this is the case. The simulations revealed that the MRd-CplD latency distribution achieved by simulating 15000 MRd requests to the RC, fits that of the sampled data. The ideally drawn latency samples modeled by the RC TLM are approximately equal to the MRd-CplD delays that are experienced by endpoints. Slight changes in the nano-second scale are caused by imperative switch arbitration cycles.

Introduction of system jitter caused by The RC TLM include jitter that is introduced by multiple performance degrading traffic scenarios. Some of these are:

- Flow control halts.
- Cache snooping halts.
- Strict ordering halts.
- Violations to the read completion boundary.
- Violations to the maximum payload size.

The traced PCIe system experience all these performance degrading events while running normal applications. Jitter variations with respect to the items above are thus automatically included in the traced MRd-CplD delays of the RC TLM.

The principle of modeling delays in an RC TLM using latency databases, was found to be a good alternative to the constant delay nature of the QEMU test-environment. The PCIe RC TLM will, with the appropriate trace calibrations, perform optimal with respect to modeled latency and jitter in a simulation environment. The implemented transaction level model of the PCIe root complex

enables PCIe ASIC designers to pinpoint optimal system parameters faster and more accurately than before, leading to a reduction in development cost, silicon area and power consumption.

Features such as credit flow control, TLP hints, and ways to simulate details of transaction scenarios in a more fine grained manner, remains to be implemented. Including these features in the RC TLM will improve the model's ability of QoS and error based performance simulations. Being able to draw a picture of QoS and functional correctness of a device is of great importance for PCIe. These model additions deserve to be investigated further in future work.



# References

- [1] Accellera. The systemc library. <http://www.accellera.org/downloads/standards/systemc>. Accessed:12.06.2014.
- [2] Jasmin Ajanovic. Pci express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.
- [3] Don Anderson, Tom Shanley, and Ravi Budruk. *PCI express system architecture*. Addison-Wesley Professional, 2004.
- [4] Fabrice Bellard. A new formula to compute the nth binary digit of  $\pi$ . *Formerly available online*, 1997.
- [5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
- [6] Ajay V Bhatt. Creating a pci express interconnect. URL [www.pcisig.com/specifications/pciexpress/technical\\_library/pciexpress\\_whitepaper.pdf](http://www.pcisig.com/specifications/pciexpress/technical_library/pciexpress_whitepaper.pdf), 2002.
- [7] Cisco. Internet of everything. <http://www.bloomberg.com/news/2014-01-08/cisco-ceo-pegs-internet-of-things-as-19-trillion-market.html>. Accessed:12.06.2014.
- [8] K. Creta and S. Muthrasanalluar. Opportunistic read completion combining, October 14 2004. US Patent App. 10/404,982.
- [9] Carlo Demichelis and Philip Chimento. Ip packet delay variation metric for ip performance metrics (ippm). 2002.
- [10] Låte Even. Latency realistic transaction level model of a pcie root complex. <https://github.com/Pufferfish/tlm-of-a-pcie-rootcomplex-systemc.git>, 2014.
- [11] Alex Goldhammer and John Ayer Jr. Understanding performance of pci express systems. *Xilinx WP350*, Sept, 4, 2008.
- [12] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. The uncore: A modular approach to feeding the high-performance cores. *Intel Technology Journal*, 14(3), 2010.
- [13] Eugin Hyun and Kwang-Su Seong. Design and verification for pci express controller. In *Information Technology and Applications, 2005. ICITA 2005. Third International Conference on*, volume 1, pages 581–586. IEEE, 2005.
- [14] Open SystemC Initiative et al. Ieee standard systemc language reference manual. *IEEE Computer Society*, 2006.

- [15] Intel. Intel® xeon® processor e5-2600 v2 product family, the heart of a modern data center. <http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf>. Accessed:12.06.2014.
- [16] Intel. An introduction to the intel quickpath interconnect.
- [17] PLX Technology Jack Regula. Overcoming latency in pcie systems using plx.
- [18] INTEL Jasmin Ajanovic. Pci express 3.0 overview, extensions. [http://www.hotchips.org/wp-content/uploads/hc\\_archives/hc21/1\\_sun/HC21.23.1.SystemInterconnectTutorial-Epub/HC21.23.131.Ajanovic-Intel-PCIeGen3.pdf](http://www.hotchips.org/wp-content/uploads/hc_archives/hc21/1_sun/HC21.23.1.SystemInterconnectTutorial-Epub/HC21.23.131.Ajanovic-Intel-PCIeGen3.pdf). Accessed:12.06.2014.
- [19] Mark Suresh Joshi. *C++ design patterns and derivatives pricing*, volume 2. Cambridge University Press, 2008.
- [20] Teledyne Lecroy. Manual for the lecroy pcie trace hw & sw. [http://cdn.teledynelecroy.com/files/manuals/petracertrainer\\_emluser\\_manual.pdf](http://cdn.teledynelecroy.com/files/manuals/petracertrainer_emluser_manual.pdf). Accessed:12.06.2014.
- [21] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.
- [22] Ravi Budruk Mike Jackson. *PCI Express Technology, Comprehensive Guide to Generations 1.x, 2.x and 3.0*. MindShare, 2012.
- [23] ORACLE. Sun fire x4170 m2 server. <http://www.oracle.com/us/products/servers-storage/servers/x86/sun-fire-x4170-m2-ds-079875.pdf>. Accessed:12.06.2014.
- [24] Miguel Angel Orozco, Mario Siller, and Adán Ruiz. Modeling and performance analysis of pci express.
- [25] PCI-SIG. Optimizing pci express port performance. [http://www.pcisig.com/developers/main/training\\_materials/get\\_document?doc\\_id=a3dd52f3cbf1301175b97073032bb55ec6f683f7](http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=a3dd52f3cbf1301175b97073032bb55ec6f683f7). Accessed:12.06.2014.
- [26] PCI-SIG. Pci express generation 3, logo. [http://www.pcisig.com/specifications/pciexpress/logo\\_guidelines/](http://www.pcisig.com/specifications/pciexpress/logo_guidelines/). Accessed:12.06.2014.
- [27] Rosti Bocchio Riccobene, Scandurra. Uml modelling of systemc. <http://www.cecs.uci.edu/~papers/emsoft0405/docs05/p138.pdf>. Accessed:12.06.2014.
- [28] Gurbir Singh, Robert Safranek, Nilesh Bhagat, Rob Blankenship, Ken Creta, Debendra Das Sharma, David L Hill, David Johnson, and Robert A Maddox. The feeding of high-performance processor cores—quickpath interconnects and the new i/o hubs. *Intel Technology Journal*, 14(3), 2010.
- [29] Reginald D Smith. The dynamics of internet traffic: self-similarity, self-organization, and complex phenomena. *Advances in Complex Systems*,

- 14(06):905–949, 2011.
- [30] PLX Technology. Choosing pci express packet payload size. 2007.
- [31] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3:2, 2011.
- [32] Jiri Vodrazka and Pavel Lafata. Transmission delay modeling of packet communication over digital subscriber line. *Advances in Electrical & Electronic Engineering*, 11(4), 2013.
- [33] Myers Walpole, Myers and Ye. *Probability & Statistics for Engineers & Scientists, Eighth Edition*. Pearson Education, 2007.
- [34] Benjy Weinberger, Craig Silverstein, Gregory Eitzmann, Mark Mentovai, and Tashana Landray. Google c++ style guide. *Section: Line Length*. url: [http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Line\\_Length](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Line_Length), 2008.
- [35] Root complex design checklist. [http://www.pcisig.com/developers/compliance\\_program/prod\\_listreq/PCIeRootComplexChecklist1.1.doc](http://www.pcisig.com/developers/compliance_program/prod_listreq/PCIeRootComplexChecklist1.1.doc). Accessed:12.06.2014.
- [36] Qemu homepage. <http://wiki.qemu.org/>. Accessed:12.06.2014.
- [37] Qemu software contribution. <http://wiki.QEMU.org/Contribute/SubmitAPatch>. Accessed:12.06.2014.
- [38] Third pci-express slot on high end z77 motherboards can only be utilized with an ivy bridge processor.
- [39] Open fabrics enterprise distribution(opef), test applications for packet tracing. [http://www.compsci.wm.edu/SciClone/documentation/software/communication/Mellanox/OFED-1.5.3/MLNX\\_OFED\\_LINUX-1.5.3-3.1.0-rhel6.2-x86\\_64/docs/release\\_notes/qperf\\_release\\_notes.txt](http://www.compsci.wm.edu/SciClone/documentation/software/communication/Mellanox/OFED-1.5.3/MLNX_OFED_LINUX-1.5.3-3.1.0-rhel6.2-x86_64/docs/release_notes/qperf_release_notes.txt). Accessed:12.06.2014.
- [40] Pci-sig engineering change notice, extended tags. [http://www.pcisig.com/specifications/pciexpress/specifications/ECN\\_Extended\\_Tag\\_Enable\\_Default\\_05Sept2008\\_final.pdf](http://www.pcisig.com/specifications/pciexpress/specifications/ECN_Extended_Tag_Enable_Default_05Sept2008_final.pdf). Accessed:12.06.2014.
- [41] Qemu new chipset support, intel's q35. <http://www.linux-kvm.org/wiki/images/0/06/2012-forum-Q35.pdf>. Accessed:12.06.2014.





## Acronyms

- ACK** Acknowledged - Packet for acknowledging data transfers
- AGP** Accelerated Graphics Port- Graphics card p2p channel
- ASIC** Application Specific Integrated Circuit - Circuit chip
- B2B** Back to Back - Burst sending of packets
- BAR** Base Address Register - Storing function configuration info
- CPU** Central Processing Unit- Computing component
- CRC** Cyclic Redundancy Check- Error Check
- Cpl** Completion - Packet type, responds to a request
- CplD** Completion with Data - Packet type, responds with data to a request
- DLLP** Data Link Layer Packet - Unit of data transfer for PCIe
- DMA** Direct Memory Access - Data Accessing
- DW** Double Word - Unit of 32 bits
- ECRC** End-To-End CRC - Error Check
- EISA** Extended ISA - Extension of the established ISA
- FIFO** First in First Out - Data accessing
- FMT** Format - Packet description
- FSB** Front Side Bus - Interconnect between north bridge and CPU
- GT/s** Giga Transfers per second- Measurement of transfer rate
- Gen** Generation - Used for PCIe hardware release specification.
- HW** Hardware - The tangible logic in a electronic system
- ISA** Industry Standard Architecture - Old bus standard by IBM
- IOH** I/O Hub - An Intel chipset architecture.
- MPS** Maximum Payload Size - The limit for packet payload in a system
- MRD** Memory Read - Packet request for reading data
- MW<sub>r</sub>** Memory Write - Packet request type for writing data
- NAK** Negative Acknowledgment - Packet for acknowledging data transfers
- QEMU** Quick EMUlator - SW for emulating CPUs
- QoS** Quality of Service - Deterministic latency and bandwidth
- QPI** Quick Path Interconnect - Intel's replacement for the FSB
- P2P** PCI to PCI bridge - Bridge between two PCI interconnects

- PCI** Peripheral Component Interconnect - Connection standard
- PCIe** Peripheral Component Interconnect Express - Connection standard
- RC** Root Complex - Root node of the PCIe Tree
- RCB** Read Completion Boundary - Boundary MRd-CplD payload size
- SW** Software - The non-tangible logic in an electronic system
- TLM** Transaction Level Modelling - High level model approach
- TLP** Transaction Layer Packet - Unit of data transfer for PCIe

# C++ Tool for Converting Exported Trace Files

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iomanip>

using namespace std;

int main() {
    char input_file_name[255] = "pextotext3.txt";
    char output_deltadelay_file_name[255] = "pextotext3_delta.txt";
    std::ifstream read_from_pex_file(input_file_name); // "pex_to_text_1.txt";

    remove(output_deltadelay_file_name);
    std::ofstream write_deltas_to_file(output_deltadelay_file_name);

    std::string line_string;
    std::string file_contents;

    if (write_deltas_to_file.is_open() && read_from_pex_file.is_open()) {
        //Help variables, middle_calc_registers
        double mrd_time_stamp_s = 0;
        double cpl_time_stamp_s = 0;
        double mrd_cpl_delta = 0;
        double mrd_cpl_delta_ns = 0;
        string mrd_time_stamp;
        string cpl_time_stamp;
        string tag_mrd;
        string tag_cpl;
        string time_stamp_string = "Time Stamp(";
        int index = 0;
        int index2 = 0;
        int index3 = 0;
        int mrd_counter_solutib = 0;
        int delta_counter = 0;
        bool get_next_mrd_tag=false;
        bool get_next_mrd_time_stamp=false;
        bool get_next_cpl_tag=false;
        bool get_next_cpl_time_stamp=false;
        std::vector<double> temp_mrds_vector;
        std::vector<int> temp_tags_vector;
```

```

while (std::getline(read_from_pex_file, line_string)) {
    if (line_string.find("MRd") != std::string::npos) { //Check for MRd
        get_next_mrd_tag=true;
        mrd_counter_solutib++;
    } else if (line_string.find("Tag(") != std::string::npos && ←
        get_next_mrd_tag){
        get_next_mrd_time_stamp=true;
        //Add tag to mrd_tag_vector
        tag_mrd.clear();
        index3 = line_string.find("Tag("); //Find start location of ←
            timestamp in line
        for (int i = index3 + 5; i < index3 + 7; i++) {
            tag_mrd = tag_mrd + line_string[i];
        }
        temp_tags_vector.push_back(atoi(tag_mrd.c_str())); //ADD to vector
        get_next_mrd_tag=false;
    } else if (line_string.find("Time Stamp(") != std::string::npos && ←
        get_next_mrd_time_stamp){
        //INIT vars every iteration
        mrd_time_stamp.clear();
        index = line_string.find(time_stamp_string); //9
        index2 = line_string.find("s)");
        for (int i = index + time_stamp_string.length(); i < (index2); i←
            ++){
            mrd_time_stamp = mrd_time_stamp + line_string[i];
        }
        mrd_time_stamp.erase(std::remove_if(mrd_time_stamp.begin(), ←
            mrd_time_stamp.end(), ::isspace), mrd_time_stamp.end()); //←
            remove whitespaces
        sscanf(mrd_time_stamp.c_str(), "%lf", &mrd_time_stamp_s); //←
            convert to double
        temp_mrds_vector.push_back(mrd_time_stamp_s); //ADD to vector
        get_next_mrd_time_stamp=false;
    } else if ((line_string.find("Cpl") != std::string::npos) && !(←
        temp_mrds_vector.empty())) { //Check for Cpl if MRd vector is not←
            empty
        get_next_cpl_tag=true;
    } else if ((line_string.find("Tag(") != std::string::npos) && ←
        get_next_cpl_tag){
        get_next_cpl_time_stamp=true;
        tag_cpl.clear();
        index3 = line_string.find("Tag("); //Find start location of ←
            timestamp in line
        for (int i = index3 + 5; i < index3 + 7; i++) {
            tag_cpl = tag_cpl + line_string[i];
        }
        get_next_cpl_tag=false;
    } else if ((line_string.find("Time Stamp(") != std::string::npos) && ←
        get_next_cpl_time_stamp){
        cpl_time_stamp.clear();
        index = line_string.find(time_stamp_string); //Find start location←
            of timestamp in line
        index2 = line_string.find("s)"); //Find end location of←
            timestamp in line
        for (int i = index + time_stamp_string.length(); i < (index2); i++)←
            {
            cpl_time_stamp = cpl_time_stamp + line_string[i];
        }
        cpl_time_stamp.erase(std::remove_if(cpl_time_stamp.begin(), ←
            cpl_time_stamp.end(), ::isspace), cpl_time_stamp.end()); //←
            remove whitespaces
    }
}

```

```

        sscanf(cpl_time_stamp.c_str(), "%lf", &cpl_time_stamp_s); //←
            convert to double
        //Check what timestamp to compare with, in the vector of ←
        timestamps
        //Write delta to file and remove
        for (unsigned i = 0; i < temp_tags_vector.size(); i++) {
            if (temp_tags_vector[i] == atoi(tag_cpl.c_str())) { //Current ←
                cpl tag is equal to tag_i
                //Calculate delta
                //write delta to file and reset
                mrd_cpl_delta = cpl_time_stamp_s - temp_mrds_vector[i];
                mrd_cpl_delta_ns = mrd_cpl_delta * (1000000000); //←
                    convert to nanoseconds
                write_deltas_to_file << mrd_cpl_delta_ns << endl; //←
                    Write to file
                delta_counter++;
                //Remove from temps
                temp_tags_vector.erase(temp_tags_vector.begin() + i);
                temp_mrds_vector.erase(temp_mrds_vector.begin() + i);
                break;
            }
        }
        get_next_cpl_time_stamp=false;
    }
}
cout << "Mrd_correct: " << mrd_counter_solutib << endl;
cout << "total delta is: " << delta_counter << endl;
cout << "tags vlength" << temp_tags_vector.size() << endl;
cout << "Mrds vlength" << temp_mrds_vector.size() << endl;
} else {
    cout << "Unable to open files";
}
read_from_pex_file.close();
write_deltas_to_file.close();

//Read them in again and sort them in a vector
std::ifstream read_deltas_from_file(output_deltadelay_file_name);
std::vector<int> numbers;
while (std::getline(read_deltas_from_file, line_string)) {
    numbers.push_back(atoi(line_string.c_str()));
}
cout << numbers[0] << endl;
std::sort(numbers.begin(), numbers.end());
cout << numbers[0] << endl;
read_deltas_from_file.close();
//Write them back to file sorted
std::ofstream write_sorted_deltas_to_file(output_deltadelay_file_name);
//calculate median and mean also
int sample_number = int(numbers.size());
bool even_sample_number = false;
int median = numbers[0]; //init median
int mean = 0;
int sum_all = 0;
if (sample_number % 2 == 0) {
    even_sample_number = true;
}
for (int i = 0; i < sample_number; i++) {
    sum_all += numbers[i];

    if (even_sample_number) {
        if (i == (sample_number / 2)) {
            median = (numbers[i] + numbers[i + 1]) / 2;

```

```
    }
  } else {
    if (i == ((sample_number - 1) / 2)) {
      median = (numbers[i]);
    }
  }
  write_sorted_deltas_to_file << numbers[i] << endl;

}
mean = sum_all / sample_number;
cout << "Median is equal to: " << median << endl;
cout << "Mean is equal to: " << mean << endl;
write_sorted_deltas_to_file.close();
}
```

# Python Script for Plotting of Latency Distributions

```

import numpy as NP
from matplotlib import pyplot as PLT
import time

with open('trace_data/delay_traces.txt') as f0:
    v0 = NP.loadtxt(f0, delimiter=",", dtype='int', comments="#", skiprows=0, ←
        usecols=None)
with open('generated_logs/ep_mrd_cpl_deltas.txt') as f:
    v = NP.loadtxt(f, delimiter=",", dtype='int', comments="#", skiprows=0, ←
        usecols=None)
with open('generated_logs/rc_rand_drawn_deltas.txt') as f2:
    v2 = NP.loadtxt(f2, delimiter=",", dtype='int', comments="#", skiprows=0, ←
        usecols=None)

v0_hist= NP.ravel(v0)
v0_hist.sort()
v_hist= NP.ravel(v)
v_hist.sort()
v2_hist=NP.ravel(v2)
v2_hist.sort()

fig = PLT.figure(num=None, figsize=(16, 9), dpi=80, facecolor='w', edgecolor='k')

ax = fig.add_subplot(111, axisbg='w')
ax0 = fig.add_subplot(311)
ax1 = fig.add_subplot(312)
ax2 = fig.add_subplot(313)
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')

ax.tick_params(labelcolor='w', top='off', bottom='off', left='off', right='off')

ax0.set_yscale('log')
ax1.set_yscale('log')
ax2.set_yscale('log')

fig.suptitle('Simulated MRd-CplD latency distribution ', fontsize=20)
ax0.set_title('Sampled MRd-CPID latency distribution')

```

```
ax1.set_title('Latency distribution experienced by EP')
ax2.set_title('Latency distribution modelled by RC')

ax.set_xlabel('Delay (ns)', fontsize=18)
ax.set_ylabel('Log scaled sample-number', fontsize=16)

n, bins, patches = ax0.hist(v0_hist, bins = range(0,10000,10), facecolor='green')
n, bins, patches = ax1.hist(v_hist, bins = range(0,10000,10), facecolor='green')
n, bins, patches = ax2.hist(v2_hist, bins = range(0,10000,10), facecolor='green')

fig.savefig('generated_logs/out_deltas.jpg', dpi=fig.dpi)
```



## PCIe Traffic Trace Summaries

## Gen 1, x16

**D.1 Summary of Trace Iteration 1**

Type	Total
Packets	13603679
Link Transactions	4178147
Split Transactions	1530402

**Packets**

Type	Upstream	Downstream	Total
TLP	2354116	1824031	4178147
DLLP	1283464	5777294	7060758
TS1 Ordered Set	0	0	0
TS2 Ordered Set	0	0	0
Fast Training Sequence	0	0	0
Electrical Idle Ordered Set	0	0	0
SKP Ordered Set	1137389	1227385	2364774
Compliance Pattern	0	0	0
Electrical Idle Exit Ordered Set	0	0	0
Link Event	0	0	0
Start Data Stream Ordered Set	0	0	0
End Bad Framing Token	0	0	0
End Data Stream Framing Token	0	0	0
Invalid	0	0	0
			<b>13603679</b>

**Packets.TLP**

Type	Upstream	Downstream	Total
Invalid TLP encoding	0	0	0
Memory Read (32 bit)	1530402	0	1530402
Memory Read (32 bit) - Locked	0	0	0
Memory Write (32 bit)	823714	0	823714
Memory Read (64 bit)	0	0	0
Memory Read (64 bit) - Locked	0	0	0
Memory Write (64 bit)	0	106610	106610
I/O Read Request	0	0	0
I/O Write Request	0	0	0
Configuration Read Type 0	0	0	0
Configuration Write Type 0	0	0	0
Configuration Read Type 1	0	0	0
Configuration Write Type 1	0	0	0
Message	0	0	0
Message with Data	0	0	0
Message for Advanced Switching	0	0	0
Message for Advanced Switching with Data	0	0	0
Completion	0	0	0
Completion with Data	0	1717421	1717421
Completion for Locked Memory Read	0	0	0
Completion for Locked Memory Read with Data	0	0	0
AtomicOp Fetch and Add (32 bit)	0	0	0
AtomicOp Fetch and Add (64 bit)	0	0	0
AtomicOp Unconditional Swap (32 bit)	0	0	0
AtomicOp Unconditional Swap (64 bit)	0	0	0
AtomicOp Compare and Swap (32 bit)	0	0	0
AtomicOp Compare and Swap (64 bit)	0	0	0
			<b>4178147</b>

**Packets.DLLP**

Type	Upstream	Downstream	Total
Ack	821638	1232440	2054078
Nak	0	0	0

PM	0	0	0
Vendor	0	0	0
InitFC1-P	0	0	0
InitFC1-NP	0	0	0
InitFC1-Cpl	0	0	0
UpdateFC-P	284005	2241270	2525275
UpdateFC-NP	177821	2303584	2481405
UpdateFC-Cpl	0	0	0
MRIOV	0	0	0
InitFC2-P	0	0	0
InitFC2-NP	0	0	0
InitFC2-Cpl	0	0	0
Invalid DLLP encoding	0	0	0
			<b>7060758</b>

## Link Transactions

Link acknowledge	Total
Implicit	2124064
Explicit	2054078
Incomplete	5
	<b>4178147</b>

## Split Transactions

Completion Type	Total
Successful Completion	1530399
Unsupported Request	0
Cfg Request Retry	0
Completer Abort	0
Incomplete	3
	<b>1530402</b>

## Packets.TLP.Requesters

Requester ID	Upstream	Downstream	Total
000:00:0	0	106610	106610
160:00:0	2354116	1717421	4071537

## Packets.TLP.Completers

Completer ID	Upstream	Downstream	Total
000:00:0	0	512612	512612
000:31:7	0	1204809	1204809

## Packets.TLP.Traffic Class

Traffic Class	Upstream	Downstream	Total
0	2354116	1824031	4178147
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

## Link Transactions.VC ID

Virtual Channel	Upstream	Downstream	Total
0	2354116	1824031	4178147
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

5	0	0	0
6	0	0	0
7	0	0	0

### Packets.DLLP.Flow Control

Virtual Channel	Upstream	Downstream	Total
0	461826	4544854	5006680
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

### Split Transactions.Split Trans. Performance

Requester -> Completer	Total	# LinkTras (Min)	# LinkTras (Avrg)	# LinkTras (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)
160:00:0 -> 000:31:7	1018930	2	2.18	5	296.000 ns	733.790 ns	10.156 us
160:00:0 -> 000:00:0	511472	2	2.00	3	208.000 ns	353.220 ns	9.684 us
	<b>1530402</b>						

### Split Transactions.Split Trans. Performance.Read Requests

Requester -> Completer, Reads	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
160:00:0 -> 000:31:7, Mem TC0	1018930	0.189	77.387	663.426	296.000 ns	733.790 ns	10.156 us	272.000 ns	404.440 ns	9.688 us
160:00:0 -> 000:00:0, Mem TC0	511472	3.216	95.909	508.626	208.000 ns	353.220 ns	9.684 us	184.000 ns	326.880 ns	9.464 us
	<b>1530402</b>									

### Split Transactions.Split Trans. Performance.Write Requests

Requester -> Completer, Writes	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
	<b>0</b>									

### Link Transactions.Link Trans. Performance

Transaction Type	Total	# Packets (Min)	# Packets (Avrg)	# Packets (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Pld. Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)
MW(64)	106610	1	2.00	2	8.000 ns	545.730 ns	612.000 ns	8	9.15	64
MRd(32)	1530402	1	1.60	2	8.000 ns	108.850 ns	248.000 ns	0	0.00	0
CpID	1717416	1	1.42	2	8.000 ns	196.110 ns	624.000 ns	2	32.79	256
MW(32)	823714	1	1.38	2	8.000 ns	89.540 ns	264.000 ns	4	52.99	256
	<b>4178142</b>									

### Link Transactions.Link Trans. Performance.Memory Writes

Requester, TC	Total	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Pld. Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)
000:00:0, TC0	106610	8.000 ns	545.730 ns	612.000 ns	8	9.15	64	12.466	20.603	2543.132
160:00:0, TC0	823714	8.000 ns	89.540 ns	264.000 ns	4	52.99	256	16.731	1141.907	3390.842

930324

## AHCI Transactions

### Errors.AHCI

#### Errors

Type	Upstream	Downstream	Total
Invalid Code	0	0	0
Running Disparity Error	0	0	0
Unexpected K/D Code	0	0	0
Idle Data Error (not D0.0)	0	0	0
Skip Late	0	0	0
Skew Error	0	0	0
Bad Packet Length	0	0	0
Ordered Set Format Error	0	0	0
Delimiter Error	0	0	0
Alignment Error	0	0	0
DLLP: Invalid Encoding	0	0	0
DLLP: Bad CRC16	0	0	0
DLLP: Reserved Field not 0	0	0	0
DLLP: FC Initialization Error	0	0	0
TLP: Invalid Encoding	0	0	0
TLP: Bad LCRC	0	0	0
TLP: Bad ECRC	0	0	0
TLP: Reserved Field not 0	0	0	0
TLP: Payload/Length Error	0	0	0
TLP: Length Error (not 1)	0	0	0
TLP: TC Error (not 0)	0	0	0
TLP: Attr Error (not 0)	0	0	0
TLP: AT Error (not 0)	0	0	0
TLP: Byte Enables Violation	0	0	0
Memory TLP: Address/Length Crosses 4K	0	0	0
Mem64 TLP: Used Incorrectly	0	0	0
Cfg TLP: Register Error	0	0	0
Msg TLP: Invalid Routing	0	0	0
Gen3 TLP: Bad Len CRC/Parity	0	0	0
Invalid Packet	0	0	0
FC: Invalid Advertisement	0	0	0
FC: Insufficient Credits	0	0	0
Training Sequence Format Error	0	0	0
Training Sequence Parity Error	0	0	0

## Gen 1, x16

Type	Total
Packets	14334128
Link Transactions	5937842
Split Transactions	2143197

## D.2 Summary of Trace Iteration 2

## Packets

Type	Upstream	Downstream	Total
TLP	3308019	2629823	5937842
DLLP	1531476	5435865	6967341
TS1 Ordered Set	0	0	0
TS2 Ordered Set	0	0	0
Fast Training Sequence	0	0	0
Electrical Idle Ordered Set	0	0	0
SKP Ordered Set	687282	741663	1428945
Compliance Pattern	0	0	0
Electrical Idle Exit Ordered Set	0	0	0
Link Event	0	0	0
Start Data Stream Ordered Set	0	0	0
End Bad Framing Token	0	0	0
End Data Stream Framing Token	0	0	0
Invalid	0	0	0
			<b>14334128</b>

## Packets.TLP

Type	Upstream	Downstream	Total
Invalid TLP encoding	0	0	0
Memory Read (32 bit)	2143197	0	2143197
Memory Read (32 bit) - Locked	0	0	0
Memory Write (32 bit)	1164822	0	1164822
Memory Read (64 bit)	0	0	0
Memory Read (64 bit) - Locked	0	0	0
Memory Write (64 bit)	0	155455	155455
I/O Read Request	0	0	0
I/O Write Request	0	0	0
Configuration Read Type 0	0	0	0
Configuration Write Type 0	0	0	0
Configuration Read Type 1	0	0	0
Configuration Write Type 1	0	0	0
Message	0	0	0
Message with Data	0	0	0
Message for Advanced Switching	0	0	0
Message for Advanced Switching with Data	0	0	0
Completion	0	0	0
Completion with Data	0	2474368	2474368
Completion for Locked Memory Read	0	0	0
Completion for Locked Memory Read with Data	0	0	0
AtomicOp Fetch and Add (32 bit)	0	0	0
AtomicOp Fetch and Add (64 bit)	0	0	0
AtomicOp Unconditional Swap (32 bit)	0	0	0
AtomicOp Unconditional Swap (64 bit)	0	0	0
AtomicOp Compare and Swap (32 bit)	0	0	0
AtomicOp Compare and Swap (64 bit)	0	0	0
			<b>5937842</b>

## Packets.DLLP

Type	Upstream	Downstream	Total
Ack	1161780	1737056	2898836
Nak	0	0	0

PM	0	0	0
Vendor	0	0	0
InitFC1-P	0	0	0
InitFC1-NP	0	0	0
InitFC1-Cpl	0	0	0
UpdateFC-P	262246	1777804	2040050
UpdateFC-NP	107450	1921005	2028455
UpdateFC-Cpl	0	0	0
MRIOV	0	0	0
InitFC2-P	0	0	0
InitFC2-NP	0	0	0
InitFC2-Cpl	0	0	0
Invalid DLLP encoding	0	0	0
			<b>6967341</b>

## Link Transactions

Link acknowledge	Total
Implicit	3039007
Explicit	2898835
Incomplete	0
	<b>5937842</b>

## Split Transactions

Completion Type	Total
Successful Completion	2143195
Unsupported Request	0
Cfg Request Retry	0
Completer Abort	0
Incomplete	2
	<b>2143197</b>

## Packets.TLP.Requesters

Requester ID	Upstream	Downstream	Total
000:00:0	0	155455	155455
160:00:0	3308019	2474368	5782387

## Packets.TLP.Completers

Completer ID	Upstream	Downstream	Total
000:00:0	0	721908	721908
000:31:1	0	470097	470097
000:31:7	0	1282363	1282363

## Packets.TLP.Traffic Class

Traffic Class	Upstream	Downstream	Total
0	3308019	2629823	5937842
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

## Link Transactions.VC ID

Virtual Channel	Upstream	Downstream	Total
0	3308019	2629823	5937842
1	0	0	0
2	0	0	0
3	0	0	0

4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

### Packets.DLLP.Flow Control

Virtual Channel	Upstream	Downstream	Total
0	369696	3698809	4068505
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

### Split Transactions.Split Trans. Performance

Requester -> Completer	Total	# LinkTras (Min)	# LinkTras (Avrg)	# LinkTras (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)
160:00:0 -> 000:31:7	1282324	2	2.00	3	288.000 ns	760.360 ns	16.172 us
160:00:0 -> 000:00:0	718514	2	2.00	3	200.000 ns	366.220 ns	15.680 us
160:00:0 -> 000:31:1	142357	2	4.30	5	392.000 ns	642.510 ns	14.860 us
	<b>2143195</b>						

### Split Transactions.Split Trans. Performance.Read Requests

Requester -> Completer, Reads	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
160:00:0 -> 000:31:7, Mem TC0	1282324	0.118	32.302	331.713	288.000 ns	760.360 ns	16.172 us	264.000 ns	420.330 ns	15.760 us
160:00:0 -> 000:00:0, Mem TC0	718514	1.946	99.434	526.165	200.000 ns	366.220 ns	15.680 us	176.000 ns	337.510 ns	15.656 us
160:00:0 -> 000:31:1, Mem TC0	142357	16.429	435.569	622.808	392.000 ns	642.510 ns	14.860 us	296.000 ns	393.230 ns	14.384 us
	<b>2143195</b>									

### Split Transactions.Split Trans. Performance.Write Requests

Requester -> Completer, Writes	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
	0									

### Link Transactions.Link Trans. Performance

Transaction Type	Total	# Packets (Min)	# Packets (Avrg)	# Packets (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Pld. Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)
MRd(32)	2143197	1	1.60	2	8.000 ns	108.810 ns	248.000 ns	0	0.00	0
MWw(32)	1164822	1	1.38	2	8.000 ns	90.280 ns	264.000 ns	4	52.85	256
CpID	2474368	1	1.41	2	8.000 ns	192.110 ns	640.000 ns	2	31.98	256
MWw(64)	155455	1	1.99	2	8.000 ns	544.100 ns	640.000 ns	8	10.32	64
	<b>5937842</b>									

### Link Transactions.Link Trans. Performance.Memory Writes

					Pld.			Thrpt		
--	--	--	--	--	------	--	--	-------	--	--



Requester, TC	Total	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)	MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)
160:00:0, TC0	1164822	8.000 ns	90.280 ns	264.000 ns	4	52.85	256	16.731	1136.014	3390.842
000:00:0, TC0	155455	8.000 ns	544.100 ns	640.000 ns	8	10.32	64	11.921	27.963	2543.132
	<b>1320277</b>									

## AHCI Transactions

### Errors.AHCI

#### Errors

Type	Upstream	Downstream	Total
Invalid Code	0	0	0
Running Disparity Error	0	0	0
Unexpected K/D Code	0	0	0
Idle Data Error (not D0.0)	0	0	0
Skip Late	0	0	0
Skew Error	0	0	0
Bad Packet Length	0	0	0
Ordered Set Format Error	0	0	0
Delimiter Error	0	0	0
Alignment Error	0	0	0
DLLP: Invalid Encoding	0	0	0
DLLP: Bad CRC16	0	0	0
DLLP: Reserved Field not 0	0	0	0
DLLP: FC Initialization Error	0	0	0
TLP: Invalid Encoding	0	0	0
TLP: Bad LCRC	0	0	0
TLP: Bad ECRC	0	0	0
TLP: Reserved Field not 0	0	0	0
TLP: Payload/Length Error	0	0	0
TLP: Length Error (not 1)	0	0	0
TLP: TC Error (not 0)	0	0	0
TLP: Attr Error (not 0)	0	0	0
TLP: AT Error (not 0)	0	0	0
TLP: Byte Enables Violation	0	0	0
Memory TLP: Address/Length Crosses 4K	0	0	0
Mem64 TLP: Used Incorrectly	0	0	0
Cfg TLP: Register Error	0	0	0
Msg TLP: Invalid Routing	0	0	0
Gen3 TLP: Bad Len CRC/Parity	0	0	0
Invalid Packet	0	0	0
FC: Invalid Advertisement	0	0	0
FC: Insufficient Credits	0	0	0
Training Sequence Format Error	0	0	0
Training Sequence Parity Error	0	0	0

## Gen 1, x16

Type	Total
Packets	14308891
Link Transactions	5924618
Split Transactions	2138416

## D.3 Summary of Trace Iteration 3

## Packets

Type	Upstream	Downstream	Total
TLP	3303096	2621522	5924618
DLLP	1526990	5448663	6975653
TS1 Ordered Set	0	0	0
TS2 Ordered Set	0	0	0
Fast Training Sequence	0	0	0
Electrical Idle Ordered Set	0	0	0
SKP Ordered Set	677506	731114	1408620
Compliance Pattern	0	0	0
Electrical Idle Exit Ordered Set	0	0	0
Link Event	0	0	0
Start Data Stream Ordered Set	0	0	0
End Bad Framing Token	0	0	0
End Data Stream Framing Token	0	0	0
Invalid	0	0	0
			<b>14308891</b>

## Packets.TLP

Type	Upstream	Downstream	Total
Invalid TLP encoding	0	0	0
Memory Read (32 bit)	2138416	0	2138416
Memory Read (32 bit) - Locked	0	0	0
Memory Write (32 bit)	1164680	0	1164680
Memory Read (64 bit)	0	0	0
Memory Read (64 bit) - Locked	0	0	0
Memory Write (64 bit)	0	156238	156238
I/O Read Request	0	0	0
I/O Write Request	0	0	0
Configuration Read Type 0	0	0	0
Configuration Write Type 0	0	0	0
Configuration Read Type 1	0	0	0
Configuration Write Type 1	0	0	0
Message	0	0	0
Message with Data	0	0	0
Message for Advanced Switching	0	0	0
Message for Advanced Switching with Data	0	0	0
Completion	0	0	0
Completion with Data	0	2465284	2465284
Completion for Locked Memory Read	0	0	0
Completion for Locked Memory Read with Data	0	0	0
AtomicOp Fetch and Add (32 bit)	0	0	0
AtomicOp Fetch and Add (64 bit)	0	0	0
AtomicOp Unconditional Swap (32 bit)	0	0	0
AtomicOp Unconditional Swap (64 bit)	0	0	0
AtomicOp Compare and Swap (32 bit)	0	0	0
AtomicOp Compare and Swap (64 bit)	0	0	0
			<b>5924618</b>

## Packets.DLLP

Type	Upstream	Downstream	Total
Ack	1159615	1736206	2895821
Nak	0	0	0

PM	0	0	0
Vendor	0	0	0
InitFC1-P	0	0	0
InitFC1-NP	0	0	0
InitFC1-Cpl	0	0	0
UpdateFC-P	261453	1813820	2075273
UpdateFC-NP	105922	1898637	2004559
UpdateFC-Cpl	0	0	0
MRIOV	0	0	0
InitFC2-P	0	0	0
InitFC2-NP	0	0	0
InitFC2-Cpl	0	0	0
Invalid DLLP encoding	0	0	0
			<b>6975653</b>

## Link Transactions

Link acknowledge	Total
Implicit	3028796
Explicit	2895821
Incomplete	1
	<b>5924618</b>

## Split Transactions

Completion Type	Total
Successful Completion	2138415
Unsupported Request	0
Cfg Request Retry	0
Completer Abort	0
Incomplete	1
	<b>2138416</b>

## Packets.TLP.Requesters

Requester ID	Upstream	Downstream	Total
000:00:0	0	156238	156238
160:00:0	3303096	2465284	5768380

## Packets.TLP.Completers

Completer ID	Upstream	Downstream	Total
000:00:0	0	720999	720999
000:31:1	0	465235	465235
000:31:7	0	1279050	1279050

## Packets.TLP.Traffic Class

Traffic Class	Upstream	Downstream	Total
0	3303096	2621522	5924618
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

## Link Transactions.VC ID

Virtual Channel	Upstream	Downstream	Total
0	3303096	2621522	5924618
1	0	0	0
2	0	0	0
3	0	0	0

4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

### Packets.DLLP.Flow Control

Virtual Channel	Upstream	Downstream	Total
0	367375	3712457	4079832
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

### Split Transactions.Split Trans. Performance

Requester -> Completer	Total	# LinkTras (Min)	# LinkTras (Avrg)	# LinkTras (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)
160:00:0 -> 000:00:0	717354	2	2.01	3	200.000 ns	390.380 ns	9.988 us
160:00:0 -> 000:31:7	1279050	2	2.00	2	288.000 ns	786.430 ns	10.428 us
160:00:0 -> 000:31:1	142012	2	4.28	5	408.000 ns	618.350 ns	10.092 us
	<b>2138416</b>						

### Split Transactions.Split Trans. Performance.Read Requests

Requester -> Completer, Reads	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
160:00:0 -> 000:00:0, Mem TC0	717354	3.127	95.673	508.626	200.000 ns	390.380 ns	9.988 us	176.000 ns	361.160 ns	9.864 us
160:00:0 -> 000:31:7, Mem TC0	1279050	0.183	31.726	190.735	288.000 ns	786.430 ns	10.428 us	272.000 ns	441.800 ns	9.880 us
160:00:0 -> 000:31:1, Mem TC0	142012	24.192	467.052	598.384	408.000 ns	618.350 ns	10.092 us	304.000 ns	415.290 ns	9.632 us
	<b>2138416</b>									

### Split Transactions.Split Trans. Performance.Write Requests

Requester -> Completer, Writes	Total	Thrpt MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Latency (Min)	Latency (Avrg)	Latency (Max)
	0									

### Link Transactions.Link Trans. Performance

Transaction Type	Total	# Packets (Min)	# Packets (Avrg)	# Packets (Max)	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Pld. Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)
MRd(32)	2138416	1	1.60	2	8.000 ns	109.020 ns	248.000 ns	0	0.00	0
CpID	2465283	1	1.41	2	8.000 ns	192.180 ns	640.000 ns	2	32.05	256
MWr(64)	156238	1	1.99	2	8.000 ns	543.680 ns	624.000 ns	8	10.57	64
MWr(32)	1164680	1	1.39	2	8.000 ns	90.540 ns	248.000 ns	4	52.83	256
	<b>5924617</b>									

### Link Transactions.Link Trans. Performance.Memory Writes

					Pld.			Thrpt		

Requester, TC	Total	Resp. time (Min)	Resp. time (Avrg)	Resp. time (Max)	Bytes (Min)	Pld. Bytes (Avrg)	Pld. Bytes (Max)	MB/s (Min)	Thrpt MB/s (Avrg)	Thrpt MB/s (Max)
000:00:0, TC0	156238	8.000 ns	543.680 ns	624.000 ns	8	10.57	64	12.227	29.601	2543.132
160:00:0, TC0	1164680	8.000 ns	90.540 ns	248.000 ns	4	52.83	256	16.731	1133.936	3390.842
	<b>1320918</b>									

## AHCI Transactions

### Errors.AHCI

#### Errors

Type	Upstream	Downstream	Total
Invalid Code	0	0	0
Running Disparity Error	0	0	0
Unexpected K/D Code	0	0	0
Idle Data Error (not D0.0)	0	0	0
Skip Late	0	0	0
Skew Error	0	0	0
Bad Packet Length	0	0	0
Ordered Set Format Error	0	0	0
Delimiter Error	0	0	0
Alignment Error	0	0	0
DLLP: Invalid Encoding	0	0	0
DLLP: Bad CRC16	0	0	0
DLLP: Reserved Field not 0	0	0	0
DLLP: FC Initialization Error	0	0	0
TLP: Invalid Encoding	0	0	0
TLP: Bad LCRC	0	0	0
TLP: Bad ECRC	0	0	0
TLP: Reserved Field not 0	0	0	0
TLP: Payload/Length Error	0	0	0
TLP: Length Error (not 1)	0	0	0
TLP: TC Error (not 0)	0	0	0
TLP: Attr Error (not 0)	0	0	0
TLP: AT Error (not 0)	0	0	0
TLP: Byte Enables Violation	0	0	0
Memory TLP: Address/Length Crosses 4K	0	0	0
Mem64 TLP: Used Incorrectly	0	0	0
Cfg TLP: Register Error	0	0	0
Msg TLP: Invalid Routing	0	0	0
Gen3 TLP: Bad Len CRC/Parity	0	0	0
Invalid Packet	0	0	0
FC: Invalid Advertisement	0	0	0
FC: Insufficient Credits	0	0	0
Training Sequence Format Error	0	0	0
Training Sequence Parity Error	0	0	0



# Simulation Output Format

## E.1 Root Complex Completes a MRd Request

```

----- INITIATING SIMULATION -----
----- SENDING_TLP -----
| PciDevice ID 200 is sending a TLP @ 0 s |
|----- TLP -----|
|----- MemReqHEADER -----|
|| (Fmt=0, Type=0, TC=5, TD=0, EP=0, Attr=11, length=10) ||
|| (RequesterID=200, Tag=0, firstDWBE=1111, lastDWBE=0001) ||
|| (Address1=25) ||
|| (Address2=-1) ||
|-----|
| [ TLP Data = < > ] |
| [ TLP ECRC = < > ] |
|-----|

----- SWITCHING_TLP_UPSTREAM -----
| PciSwitch ID 100 is recieving a TLP @ 3 ns |
|----- TLP -----|
|----- MemReqHEADER -----|
|| (Fmt=0, Type=0, TC=5, TD=0, EP=0, Attr=11, length=10) ||
|| (RequesterID=200, Tag=0, firstDWBE=1111, lastDWBE=0001) ||
|| (Address1=25) ||
|| (Address2=-1) ||
|-----|
| [ TLP Data = < > ] |
| [ TLP ECRC = < > ] |
|-----|

----- SWITCHING_TLP_UPSTREAM -----
| PciSwitch ID 000 is recieving a TLP @ 6 ns |
|----- TLP -----|
|----- MemReqHEADER -----|
|| (Fmt=0, Type=0, TC=5, TD=0, EP=0, Attr=11, length=10) ||
|| (RequesterID=200, Tag=0, firstDWBE=1111, lastDWBE=0001) ||
|| (Address1=25) ||
|| (Address2=-1) ||
|-----|

```

```

| [ TLP Data = < > ]
| [ TLP ECRC = < > ]
|-----|
|
|-----RECIEIVING_TLP-----|
| ROOT COMPLEX is recieving a TLP @ 6 ns
|-----TLP-----|
|-----MemReqHEADER-----|
| (Fmt=0, Type=0, TC=5, TD=0, EP=0, Attr=11, length=10)
| (RequesterID=200, Tag=0, firstDWBE=1111, lastDWBE=0001)
| (Address1=25)
| (Address2=-1)
|-----|
| [ TLP Data = < > ]
| [ TLP ECRC = < > ]
|-----|
|
|//////////////////// COMMENCING MEMORYREADING //////////////////////|
|---DW at address:25, that is: 25. Is available on the DATAFromMemory bus
|---DW at address:26, that is: 26. Is available on the DATAFromMemory bus
|---DW at address:27, that is: 27. Is available on the DATAFromMemory bus
|---DW at address:28, that is: 28. Is available on the DATAFromMemory bus
|---DW at address:29, that is: 29. Is available on the DATAFromMemory bus
|---DW at address:30, that is: 30. Is available on the DATAFromMemory bus
|---DW at address:31, that is: 31. Is available on the DATAFromMemory bus
|---DW at address:32, that is: 32. Is available on the DATAFromMemory bus
|---DW at address:33, that is: 33. Is available on the DATAFromMemory bus
|---DW at address:34, that is: 34. Is available on the DATAFromMemory bus
|//////////////////// DONE WITH MEMORYREADING //////////////////////|
|
|-----SENDING_TLP-----|
| ROOT COMPLEX is sending a TLP @ 328 ns
|-----TLP-----|
|-----CplHEADER-----|
| (Fmt=0, Type=12, TC=5, TD=0, EP=0, Attr=00, length=10)
| (Completer ID=000, ComplStatus=0, BCM=0, byteCount=0)
| (Requester ID=200, Tag=0, LowerAddress=0)
|-----|
| [ TLP Data = < 25 26 27 28 29 30 31 32 33 34 > ]
| [ TLP ECRC = < > ]
|-----|
|
|-----SWITCHING_TLP_DOWNSTREAM-----|
| PciSwitch ID 000 forwards a packet to bus nr:2 @ 328 ns
|-----TLP-----|
|-----CplHEADER-----|
| (Fmt=0, Type=12, TC=5, TD=0, EP=0, Attr=00, length=10)
| (Completer ID=000, ComplStatus=0, BCM=0, byteCount=0)
| (Requester ID=200, Tag=0, LowerAddress=0)
|-----|
| [ TLP Data = < 25 26 27 28 29 30 31 32 33 34 > ]
| [ TLP ECRC = < > ]
|-----|
|
|-----SWITCHING_TLP_DOWNSTREAM-----|
| PciSwitch ID 100 forwards a packet to bus nr:2 @ 328 ns

```



```

-----TLP-----
-----CplHEADER-----
| (Fmt=0, Type=12, TC=5, TD=0, EP=0, Attr=00, length=10) |
| (Completer ID=000, ComplStatus=0, BCM=0, byteCount=0) |
| (Requester ID=200, Tag=0, LowerAddress=0) |
-----
| [ TLP Data = < 25 26 27 28 29 30 31 32 33 34 > ] |
| [ TLP ECRC = < > ] |
-----

-----RECIEVING_TLP-----
PCIdevice ID 200 is recieving a TLP @ 328 ns
-----TLP-----
-----CplHEADER-----
| (Fmt=0, Type=12, TC=5, TD=0, EP=0, Attr=00, length=10) |
| (Completer ID=000, ComplStatus=0, BCM=0, byteCount=0) |
| (Requester ID=200, Tag=0, LowerAddress=0) |
-----
| [ TLP Data = < 25 26 27 28 29 30 31 32 33 34 > ] |
| [ TLP ECRC = < > ] |
-----

PCI-Socket: Removing outbound request @328 ns
--ENDPOINT 200 receives: EP has now a total of 1 Packets @ 328 ns

----- SIMULATION COMPLETE -----

-----
                                @10 us
-----

Program was executed with the following parameters:
./PCIEsystem.out 1 0 0 MRd 0 MRd 10000 -show_full -out.txt

-----
Simulation statistics:
Format: EPs and RC <req_sent/cpl_rec><req_rec/cpl_sent>
Sockets: <sent/received>, Switches, <rec_ds/fwd_us><rec_us/fwd_ds>

-----
Packet Statistics: <0/0> <1/1>
Requests received time stamps:
        6 ns
Completions sent time stamps:
        328 ns
--Switch: 000 Destructor--
Packet Statistics: <1/1> <1/1>
--Socket Destructor--
Packet Statistics: <1/1>

--EndPoint: 200 Destructor--
Packet Statistics: <1/1> <0/0>
Requests sent time stamps:
        0 s
Completions received time stamps:

```

```
328 ns
Average MRd-CplD time is:
328000 ps
Data received is:
25 26 27 28 29
30 31 32 33 34

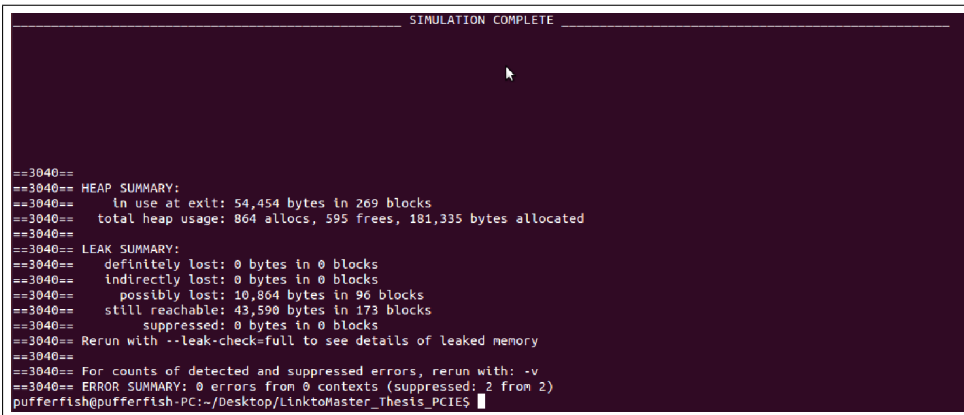
--Socket Destructor--
Packet Statistics: <1/1>

--Switch: 100 Destructor--
Packet Statistics: <1/1> <1/1>

- - - The PCIe System has been destroyed - - -
```

## Valgrind Test for Program Runability

It was discovered during Valgrind memory checks that the Valgrind tool does not support the standard compiled SystemC library. Valgrind reports a high amounts of definitely lost memory due to the way the concurrent threads are defined in the SystemC library. In order to fix this problem for a correct memory-check run, the SystemC library had to be recompiled using the pthreads flag.



```
SIMULATION COMPLETE

==3040==
==3040== HEAP SUMMARY:
==3040==   in use at exit: 54,454 bytes in 269 blocks
==3040==   total heap usage: 864 allocs, 595 frees, 181,335 bytes allocated
==3040==
==3040== LEAK SUMMARY:
==3040==   definitely lost: 0 bytes in 0 blocks
==3040==   indirectly lost: 0 bytes in 0 blocks
==3040==   possibly lost: 10,864 bytes in 96 blocks
==3040==   still reachable: 43,590 bytes in 173 blocks
==3040==   suppressed: 0 bytes in 0 blocks
==3040== Rerun with --leak-check=full to see details of leaked memory
==3040==
==3040== For counts of detected and suppressed errors, rerun with: -v
==3040== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
pufferfish@pufferfish-PC:~/Desktop/LinktoMaster_Thesis_PCIES
```

**Figure F.1:** Summary log for a valgrind run, no definitely lost memory

The definitely lost section of the Valgrind was greatly reduced with the workaround consisting of compiling SystemC with the Pthreads flag. The remaining leaks were located and corrected, these were mostly fixed by correcting destructor to include destruction of objects being pointed to.

