Performance Monitoring of Multi-FPGA Systems

by

Arzhang Rafii

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Performance Monitoring of Multi-FPGA Systems

Arzhang Rafii
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2021

Field-Programmable Gate Arrays (FPGAs) have been increasingly deployed in datacenters and there has been a lot of focus on tools that help the development of FPGA applications. Among the most important tools are performance monitors that provide visibility into the state of the hardware. As the application platforms scale from one FPGA to many FPGAs, such as the well-known examples demonstrated on the Microsoft Catapult platform, it is no longer feasible to rely on low-level FPGA tools such as embedded logic analyzers. This thesis presents Pharos, a lightweight, generic performance monitor for multi-FPGA systems. We show that Pharos is able to measure cycle-accurate unidirectional point-to-point latency between multiple network-connected FPGAs. Pharos is also capable of monitoring network traffic and conducting throughput measurements, as well as tracing events across the datacenter.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Paul Chow. The work presented here would not be possible without his continuous support and insightful guidance. Thank you for putting your trust in me. Your wisdom and generosity helped me become a better engineer and a better person.

I owe special thanks to some of my colleagues in Professor Chow's group. Mohammad Ewais, for his help on editing this document. Clark Shen and Camilo Vega for their countless technical supports. And Marco Merlini, who generously helped me debug my project.

To all of my wonderful friends and colleagues in Professor Chow's group, (alphabetically) Mohammad Ewais, Charles Lo, Daniel Ly-Ma, Fernando Martin del Campo, Marco Merlini, Daniel Rozhko, Varun Sharma, Clark Shen, Naif Tarafdar, and Camilo Vega. I have learned a great deal from you. Thank you for your generous support. You truly set an example of what a team should be like.

To my parents, Farhang Rafii and Maria Ilka, who left everything behind to come to Canada so I can seek a better future. No words can express how grateful I am to you. To my wonderful brother, Arya Rafii, whose emotional support helped me overcome hurdles along the way and never hesitated to lend a friendly ear. And to Maryam Naghibzadeh, who always pushed me to be better and never stopped believing in me even when I had doubts about myself. I couldn't be more grateful to you all.

Contents

Chapter 1

Introduction

In recent years, Field Programmable Gate Arrays (FPGAs) have increasingly been deployed in datacenters and cloud computing infrastructure as compute devices. Microsoft's Project Catapult [1][2] is perhaps one of the most well-known examples. Similar work has been done on deploying FPGAs in the cloud by IBM [3], Amazon [4], and other academic research [5][6][7]. FPGAs, for their programmability, offer the flexibility to optimize the hardware to use the available resources more efficiently. On the other hand, it is the responsibility of datacenter providers to ensure the datacenter operates at maximum efficiency. They are often faced with a set of challenges known as "performance debugging." Performance debugging includes answering a category of questions such as: "Why are tasks running slower than the expected speed"? "How do we distribute the current workload of all tasks within the datacenter to avoid network congestion"? "Given a new application, in which region of the cluster should it run"?

To answer such questions, it is necessary to constantly monitor the status of the datacenter. This spans from monitoring point-to-point latency to measuring throughput and bandwidth usage of different nodes within the network. Performance monitoring often proves to be a challenging task. First, there is always a trade-off between the cost and accuracy of the tools. While we would like the performance monitor to be as accurate as possible, we want to avoid adding latency to the system by the mere act of monitoring it. However, accuracy and cost (in terms of FPGA resources) normally fall on two different ends of the spectrum. Hardware-based monitors are normally more accurate, yet they tend to be more expensive in terms of using the FPGA resources. On the other hand, while software-based solutions usually do not use much of the FPGA resources, they are not as accurate. Another challenge is separating the monitor from the lower-level network protocols. Many tools offer latency measurements by timestamping certain types of packets, such as UDP or IPV4. This is undesirable since it limits the tool to be used only for a certain type of traffic.

In this thesis, we present Pharos, an open source [8] performance monitor for multi-FPGA systems. Pharos is a collection of standalone hardware modules that provide capabilities such as latency and throughput measurements and event logging. The high-level architecture of Pharos is shown in Figure 1.1. Pharos uses a modified version of the Precision Time Protocol (PTP) [9] to synchronize times of various nodes[1] within the network. Introducing a common time that all nodes agree upon gives birth to the possibility of accurate unidirectional latency measurements. Furthermore, network time synchronization enables the performance monitor to log events on multiple nodes and sort them

[1]We refer to each device in the network, in this case an FPGA, as a network node.

chronologically.

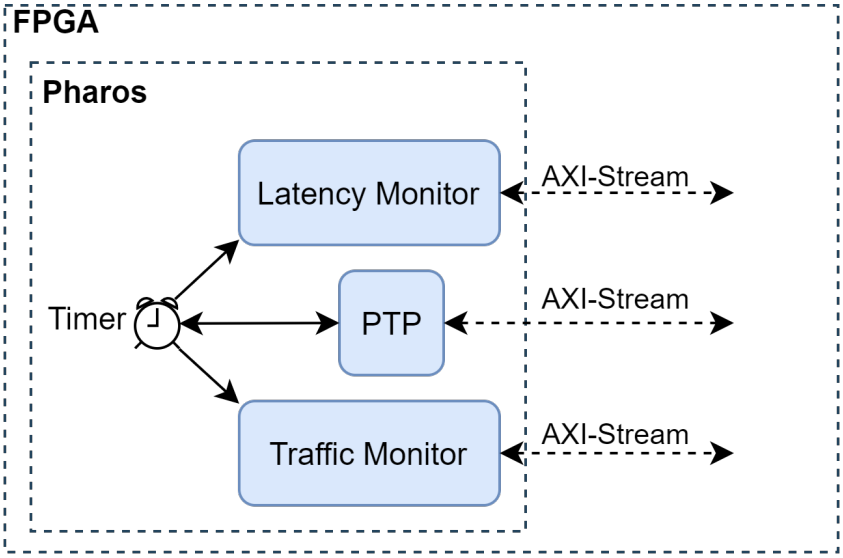

Figure 1.1: High-level architecture of Pharos

To make Pharos independent of the lower communication protocols, all of the timestamping is done within the FPGA and by using the AXI interface [10]. This means that Pharos is unaware of what communication protocol is being used between the nodes of the datacenter and does not need to place timestamps within a specific type of network packet. As long as AXI packets can be encapsulated using a lower-level network packet, Pharos can be used to monitor the performance of the datacenter. In our experiments, we have used the UDP protocol for inter-FPGA communications. However, the network protocol can be easily changed without the need to alter the performance monitor. In addition to timestamping, all of the other external communications of Pharos are also done using the AXI4-Stream protocol.

1.1 Motivation

The main motivation of this work is to enable performance monitoring and performance debugging of single- and multi-FPGA systems. Finding slow regions of the datacenter can be hugely beneficial to improving its efficiency. The main objective of the latency monitoring system of Pharos is to provide designers and datacenter administrators the information that helps them balance the traffic throughout the network. This can also be beneficial when it comes to the problem of placement. In this case, we are not referring to the placement within one FPGA, but the task of determining where to run a new application on a datacenter. The goal of placement is to identify regions of the cluster that have less traffic and avoid the nodes that are highly congested. A multi-FPGA performance monitor can provide the tools to collect such information.

In addition to performance debugging, performance monitoring can help functional debugging of applications. This can be especially beneficial when it comes to high-level debugging. Monitoring traffic and logging (and tracing) events enables designers to monitor inter-application (or at a higher level inter-FPGA) communications. Furthermore, combining event logging capabilities with the idea of global time

gives rise to event tracing capabilities. This means that certain packets can be traced throughout the network by placing event loggers at certain checkpoints. These checkpoints will then provide timestamps whenever they detect certain packets. Time synchronization enables the performance monitor to sort these timestamps in a meaningful order and provides designers with accurate information of packet trips around the network.

Finally, performance monitoring can be beneficial in the behavioural modelling of applications. Often hardware designers are faced with questions such as: "How will certain design decisions change the behaviour of my application"? Or "How does a certain change in the functionality of a hardware application affect the network traffic"? Performance monitoring provides designers with tools that can guide them to make better design decisions.

1.2 Contributions

The contribution of this thesis is Pharos: a performance monitor for multi-FPGA systems. The major components of this contribution are:

- A PTP-based time synchronization protocol for multi-FPGA systems
- A latency monitoring system capable of measuring point-to-point unidirectional latency within multi-FPGA systems
- A traffic monitoring system capable of snooping on AXI4-Stream communications and providing information about the passing traffic
- An event logging system capable of triggering on user-specified conditions and ordering events in the correct sequence within multi-FPGA systems.

1.3 Overview

The remainder of this thesis is organized as follows: Chapter 2 provides background information and related work on performance monitoring of software and hardware applications, as well as background on network time synchronization and the communication protocols used in this work. Chapter 3 explains the architecture of the Pharos performance monitor and the details of all of its components. Chapter 4 discusses the evaluation of Pharos, including methodology and results for each of the experiments. Finally, Chapter 5 concludes the thesis and discusses future works.

Chapter 2

Background and Related Work

This chapter provides background information on the main concepts used in this thesis. We start by looking into network time synchronization in datacenters and two of the most used time synchronization protocols: The Network Time Protocol (NTP) and the Precision Time Protocol (PTP). We then discuss the AXI interface, and more specifically the AXI4-Stream protocol, the main communication protocol used in the architecture of Pharos. Section 2.3 discusses the software-based performance monitors and some previous solutions to the performance monitoring problem in the software realm. Finally, Section 2.4 presents some hardware performance monitoring techniques and related work, including state-of-the-art hardware-based performance monitoring systems.

2.1 Network Time Synchronization

Time synchronization plays an essential role in many modern day network applications. Many applications highly rely on determining the sequence of events. For instance, in many financial applications such as stock trading, it is necessary to determine which client order precedes the others. Additionally, in many scientific, financial, or industrial applications it is important to know how long a certain set of tasks take to run. When these tasks are distributed among many machines, determining the sequence of events and estimating their runtime gets increasingly difficult. To overcome such problems, time synchronization algorithms are used to ensure all machines within the network agree on a common global time.

In addition to the aforementioned use cases, time synchronization can be highly beneficial in performance monitoring applications. When network nodes agree on a common time, it becomes easier to conduct latency measurements since round-trip time measurement of packets is no longer necessary. Furthermore, the global time makes it possible to sort events in a meaningful order and trace them within the network. This section describes two of the most frequently used time synchronization algorithms in modern networks: the Network Time Protocol (NTP) and the Precision Time Protocol (PTP).

2.1.1 Network Time Protocol

The Network Time Protocol (NTP) [11] is a clock synchronization protocol introduced by David L. Mills in 1991. The goal of NTP is to synchronize times of all nodes in a network within a few milliseconds. As shown in Figure 2.1, NTP uses a hierarchical structure. Each level of the hierarchy is called a *Stratum*.

Stratum 0 includes a set of high-precision clocks, such as atomic clocks, radio clocks, or GPS. The devices in Stratum 1 are connected to clocks of Stratum 0 using a direct connection (wire or radio). After the first Stratum, every Stratum synchronizes its clocks to the previous Stratum using network connections. NTP supports up to 15 Strata.

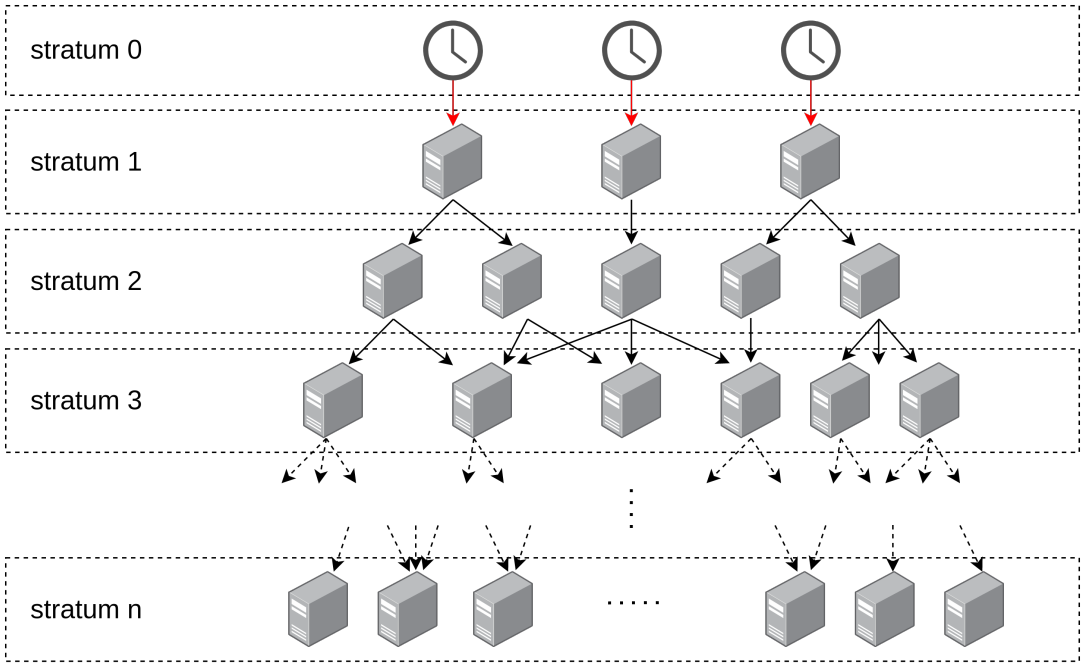

Figure 2.1: Hierarchical structure of Network Time Protocol. The red arrows indicate a direct connection, while the black arrows are typical network connections.

NTP is built on the Internet Protocol (IP) and the User Datagram Protocol (UDP). The synchronization is done using 64-bit timestamps that are sent using UDP. NTP nodes are categorized into primary servers, secondary servers, and clients. Primary servers are the nodes in Stratum 1 that are directly connected to a reference clock. All the nodes in higher Strata are secondary servers, meaning that nodes of Stratum n have a server-client relationship with the nodes in Stratum n+1. The Bellman-Ford algorithm [12][13] is used to minimize the communication delay of each node with the nodes in Stratum 1. Each node within a Stratum can communicate with multiple nodes from its immediate lower Stratum for better accuracy. Moreover, each node can talk to other nodes within the same Stratum for sanity checking.

2.1.2 Precision Time Protocol

The Precision Time Protocol (PTP) [9] is an IEEE standard used for the synchronization of clocks within heterogeneous networks. PTP supports a sub-microsecond synchronization accuracy between clocks of different stability and resolution by defining a master-slave relationship between all PTP instances. Figure 2.3 shows the sequence of messages between PTP-master and PTP-slave instances, which would typically be on different systems. The message sequence is as follows:

1. Master initializes the synchronization with a sync message at time (t_{m1})
2. Slaves records the time at which it receives the sync message (t_{s1})

3. Master sends a follow-up message with the t_{m1} timestamp
4. Slave sends a delay request and records the time at which the delay request is sent (t_{s2})
5. Master receives the delay request and immediately sends back a delay response with a new timestamp (t_{m2})

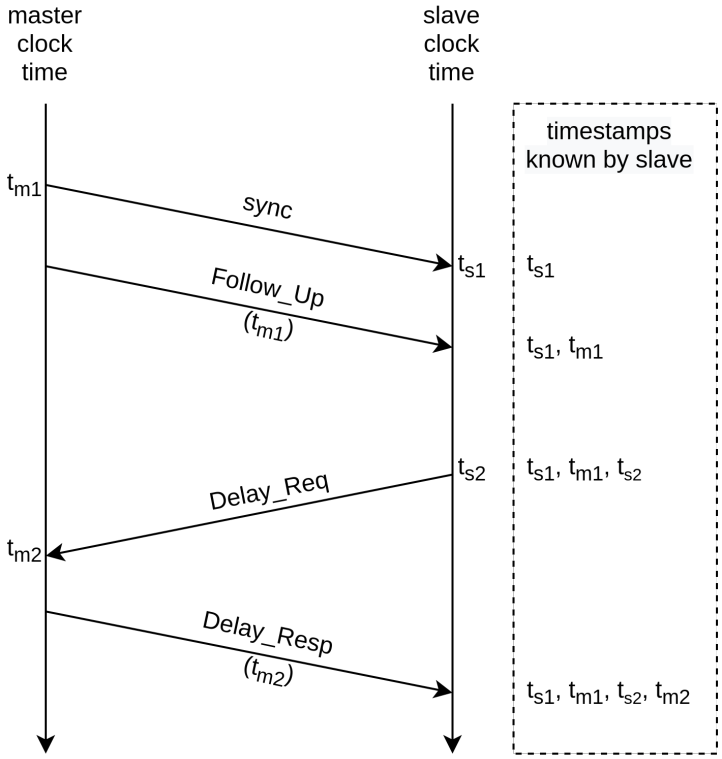

Figure 2.2: Precision Time Protocol timestamping sequence

The time-offset between the master and slave is then calculated using Equation 2.1:

$$t_{offset} = \frac{(t_{s1} - t_{m1}) - (t_{m2} - t_{s2})}{2} \tag{2.1}$$

In the offset time calculation, it is assumed that the propagation times between the master and the slave are symmetrical, e.g. master-to-slave propagation is equal to slave-to-master propagation. Therefore, any asymmetry will introduce an error to the t_{offset} calculation. The synchronization (aforementioned steps 1-5) must be repeated periodically to make sure the master and slave times do not drift apart over time. The period between every two consecutive synchronizations is referred to as the *Synchronization Interval* and denoted as T_{sync}. The synchronization interval is typically set based on the rate at which PTP clocks drift apart from each other and the amount of error that is tolerable for the datacenter's specific application.

There are several factors that affect the clock drift between nodes. One of the most important factors is the stability of the clock generation crystals. Physical properties of crystals cause a certain amount of inaccuracy in the clock frequency they generate. For example, a typical reference crystal from Renesas Electronics [14] offers clock stability with a maximum drift of 20ppm at $25°C$. This means that for every

1 million cycles that the crystal generates at $25°C$, there may be up to 20 cycles inaccuracy. Aging of the crystals normally has a negative effect on their accuracy. Temperature also plays a role in the clock generation accuracy. Typically, uncompensated crystals have a thermal inaccuracy of 1 ppm per degree Celsius [9]. In terms of PTP, this means that for a T_{sync} of 2 seconds, every $1°C$ rise in temperature causes an error on the order of 2 μs.

The PTP network is usually arranged in a hierarchical structure. In this structure, one of the clocks is chosen as the *GrandMaster* clock. This clock provides the timing reference for the whole network. The *Boundary Clocks* are connected to multiple sub-trees and can synchronize different network segments to each other. Lastly, *Ordinary Clocks* are the leaf nodes with a single network connection. The GrandMaster and the clock hierarchy is determined through the Best Master Clock Algorithm [9]. In a synchronization process, the GrandMaster clock broadcasts its time to the Boundary Clocks that it is directly connected to. The Boundary Clocks then propagate the synchronization to the rest of the network.

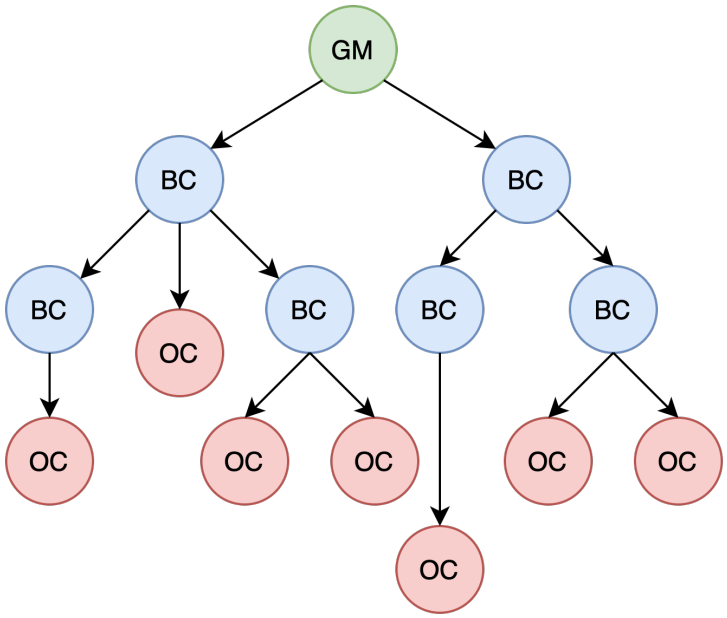

Figure 2.3: Hierarchical structure of the Precision Time Protocol. GM, BC, and OC denote GrandMaster, Boundary Clock, and Ordinary Clock, respectively.

The protocol described in this section is an implementation for commodity hardware. Many off-the-shelf network switches and routers offer PTP synchronization. Cisco 7600 Series Routers [15] and Dell EMC PowerSwitch S4100-ON Series Switches [16] are examples of PTP-aware devices. There are also many standalone chips that specifically provide PTP support. These devices are usually placed in the Ethernet PHY of various nodes within the network to access network packets as soon as they are available within the wire. Texas Instruments DP83640 [17] and Microsemi ZL30347 [18] are examples of such devices. Most PTP-aware devices and standalone PTP solutions are designed to timestamp Ethernet packets.

2.2 AXI Interface

The Advanced Extensible Interface (AXI) [10] is a communication interface that is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) [19]. AXI is widely used in industry as an on-chip communication protocol and includes different protocols for data streams and memory mapped communications. The Pharos performance monitor is built on Xilinx FPGAs and uses the Xilinx FPGA CAD tool, the Vivado Design Suite [20] [21]. Since most of the standard IPs of Vivado use AXI as their main communication protocol, Pharos is designed to use AXI as its main communication interface. This decision has other benefits that are discussed later in Chapter 3. This section provides background information about the latest version of the streaming mode of the AXI interface (AXI4-Stream), which is used in Pharos as the main communication protocol.

AXI-Stream [22] is a data streaming protocol that allows point-to-point communication by defining a master-slave relationship. AXI-stream can transfer data every cycle through a handshake between the master and the slave instances. The handshake is done using two signals: TVALID and TREADY. TVALID is a signal asserted by the master indicating a valid data transfer. On the other hand, slave uses TREADY to indicate that it is ready to accept a new data transfer. A low ready signal will cause a back-pressure on the master's data line. When both TVALID and TREADY signals are high, a valid data transfer takes place.

In addition to the TVALID and TREADY signals, AXI includes several other signals (sometimes referred to as channels). Table 2.1 lists the signals that are used in the work of this thesis. Some of the signals that are not used in the design of Pharos are excluded alongside the clock and reset signals. AXI4-Stream data is transferred in an integer number of bytes, spanning from a minimum 1-byte transfer to a maximum of 64 bytes. The TKEEP signal determines which bytes of the payload must be transmitted as part of a valid transfer. Therefore, the width of the TKEEP signal is always $1/8$ of the width of TDATA, with every bit of TKEEP indicating the validity of each byte of the payload. The data bytes with their associated keep signal deasserted are discarded from the data stream. It is important to note that TKEEP signals can be asserted to *high* consecutively only from one side of the bytes transferred. For example, for a 4-bit TKEEP, 1100 and 0011 are valid TKEEP values, and 0110 is invalid.

The AXI communications are done using AXI flits and AXI packets. The packets indicate the boundaries of data and often carry control information or payload. Each packet can be broken down to any number of flits (or flow units), which can then be sent separately through the network. The AXI4-Stream flit size is typically between 1 to 64 bytes. The packet size limit depends on the underlying network protocol that carries the AXI packets. For example, the UDP protocol allows packet sizes between 20 and 65,536 bytes. The boundary of AXI packets is indicated using the TLAST signal.

Table 2.1: Signal Description of AXI-Stream Protocol

Signal	Source	Description
TREADY	Slave	indicates that slave can receive a new transfer
TVALID	Master	indicates that the current transfer is valid
TDATA	Master	the primary payload with a width of an integer number of bytes
TKEEP	Master	byte-qualifier; indicates of which transferred bytes of TDATA must be processed as part of the data stream
TLAST	Master	indicates the end of a packet
TDEST	Master	indicates the destination address
TUSER	Master	a user-define side-channel transmitted alongside the payload

2.3 Software-based Performance Monitoring

This section discusses some of the software-based performance analysis and performance debugging techniques. We start by looking at some common manual performance debugging techniques. Section 2.3.1 discusses some of the most relevant previous solutions for software-based performance analysis.

The simplest way to analyze the performance of a software program is perhaps to use the available libraries and functions of that programming language. For instance, C/C++ libraries such as *chrono* and *time* provide the means to measure the runtime of certain sections of a program. Listing 2.1 shows an example of the usage of the C/C++ *time* library to measure the time of a function. In this case, every time the *time* function is called, a timestamp[1] is generated. In this case, *start* and *end* represent the timestamps generated at the beginning and end of the section under analysis. The *difftime* function is used to subtract the timestamps and calculate the execution time.

Time measurements offered by these libraries are easy ways to conduct runtime measurements of a software application. The precision offered by these libraries is normally sufficient for application-level analysis. The *chrono* library, for instance, offers measurements with nano-second accuracy. However, since these function calls are at the software-level, there is simply no way to eliminate the software-to-hardware overhead. This overhead is usually the time that is taken by the operating system to communicate with the CPU and kick-off/stop the timestamping.

```
#include <stdio.h>
#include <time.h>
using namespace std;

int main()
{
    time_t start, end;
    double execution_time;

    time(&start); //start measuring time
    sample_function();
    time(&end); //stop measuring time

    //calculate the execution time
    execution_time = difftime(end,start);
    return 0;
}
```

Listing 2.1: Measuring time of a code section in C/C++ using the *time* library

The aforementioned technique is only an example of time analysis in software programs. Performance monitoring generally expands to a much wider range of measurements. For single-CPU applications, performance monitoring can encompass other aspects such as memory access analysis and threading

[1]The timestamp generated by C/C++ time library represents the number of seconds passed since midnight of January 1, 1970 UTC

timelines. On the other hand, performance monitoring in datacenters is typically much more complicated. This type of monitoring usually includes (but is not limited to) measuring point-to-point latency between different nodes, event logging/tracing across the datacenter, and resource management. The rest of this section discusses some of the most relevant previous works in performance monitoring of single-CPU and multi-CPU systems.

2.3.1 Related Work

Gprof [23] is a performance monitoring tool for Unix applications. Developed in 1982, it is perhaps one of the earliest profiling tools that is still relevant today. Gprof profiles certain applications by automatically adding instrumentation code into the existing code at compile-time. Two types of analysis are provided by Gprof: flat profiling and call graph profiling. Flat profiling provides information on what routines have been called during the execution of a program, how many times they have been called, and how much time the program spent on each of them. Call graph Profiling determines the parents of a routine (the routines that called this routine) and its children (the subroutines called by that routine). Gprof determines the execution time of routines by monitoring the program counter (PC) periodically. The resolution of the profiling data is limited by this sampling rate. Also, the intrusiveness of Gprof adds runtime overhead.

Arm MAP [24] is a software profiling tool for ARM-based applications. It is designed for performance debugging of C, C++, Python, and Fortran applications. With a typical runtime overhead of about 5%, Arm MAP can be used to find bottlenecks and causes of slow performance. It provides profiling and analysis for thread activity, memory usage, I/O performance, and communications (such as MPI), as well as processor instruction-level analysis.

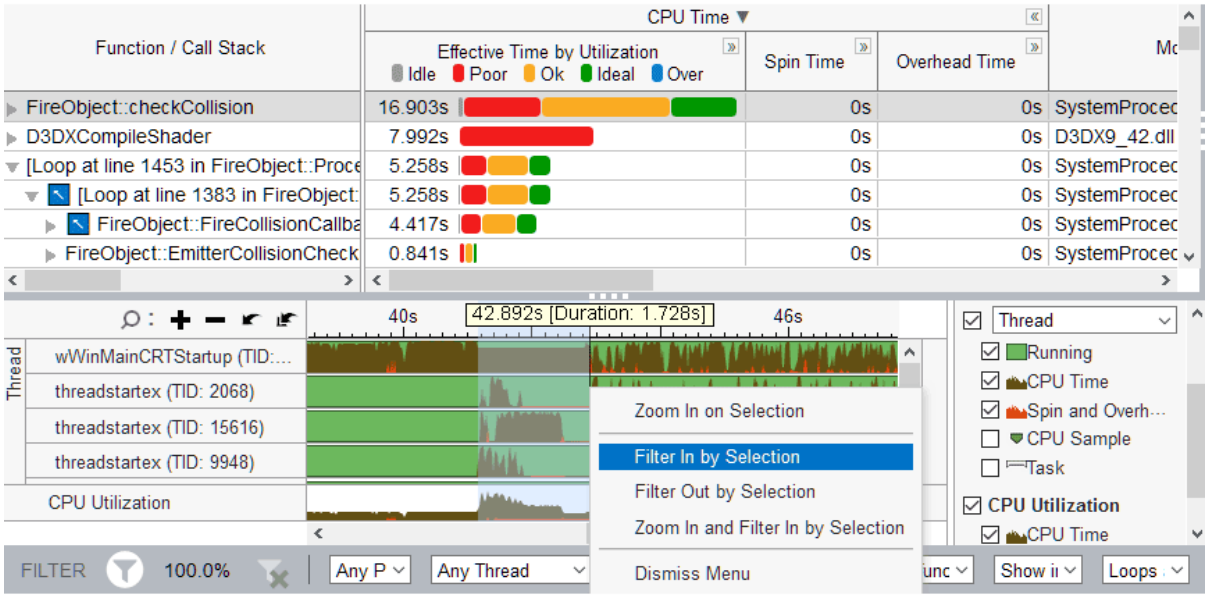

Figure 2.4: Performance analysis interface of VTune [25]

Intel's VTune [26] and AMD's CodeXL [27] are other examples of performance monitors that are

designed for application analysis on specific platforms. Intel's VTune is a performance analysis software that provides access to the hardware sampling tools of Intel processors. VTune offers single-thread and multi-thread performance analysis, as well as memory debugging. VTune also shows a system-level view of application performance. An example of program analysis by VTune is provided in Figure 2.4. AMD CodeXL is a set of open-source debugging and profiling tools for AMD GPUs and APUs. It provides debugging tools for OpenCL and OpenGL APIs and collects hardware performance counter data from AMD GPUs and APUs. It also allows time-based and event-based profiling of AMD CPUs.

There are also many examples of performance analysis tools for monitoring applications that are running on distributed systems and datacenters. MPE [28] and Datadog [29] are examples of such tools. MPI Parallel Environment (MPE) is a software package that provides profiling tools for MPI [30]. MPE includes JumpShot [28], a Java-based package that helps visualize the profiled data, such as communication between nodes, in a timeline. MPE is also an intrusive profiler, which adds run-time overhead to applications. Figure 2.5 shows an example of a performance analysis visualization using Jumphshot. Datadog is a performance monitoring and analysis tool for datacenters and cloud-scale applications. Datadog provides the ability to trace requests and events across distributed systems. It also allows monitoring of application runtimes across many machines. Datadog includes a visualization tool that helps visualize the traffic flow across the datacenter.

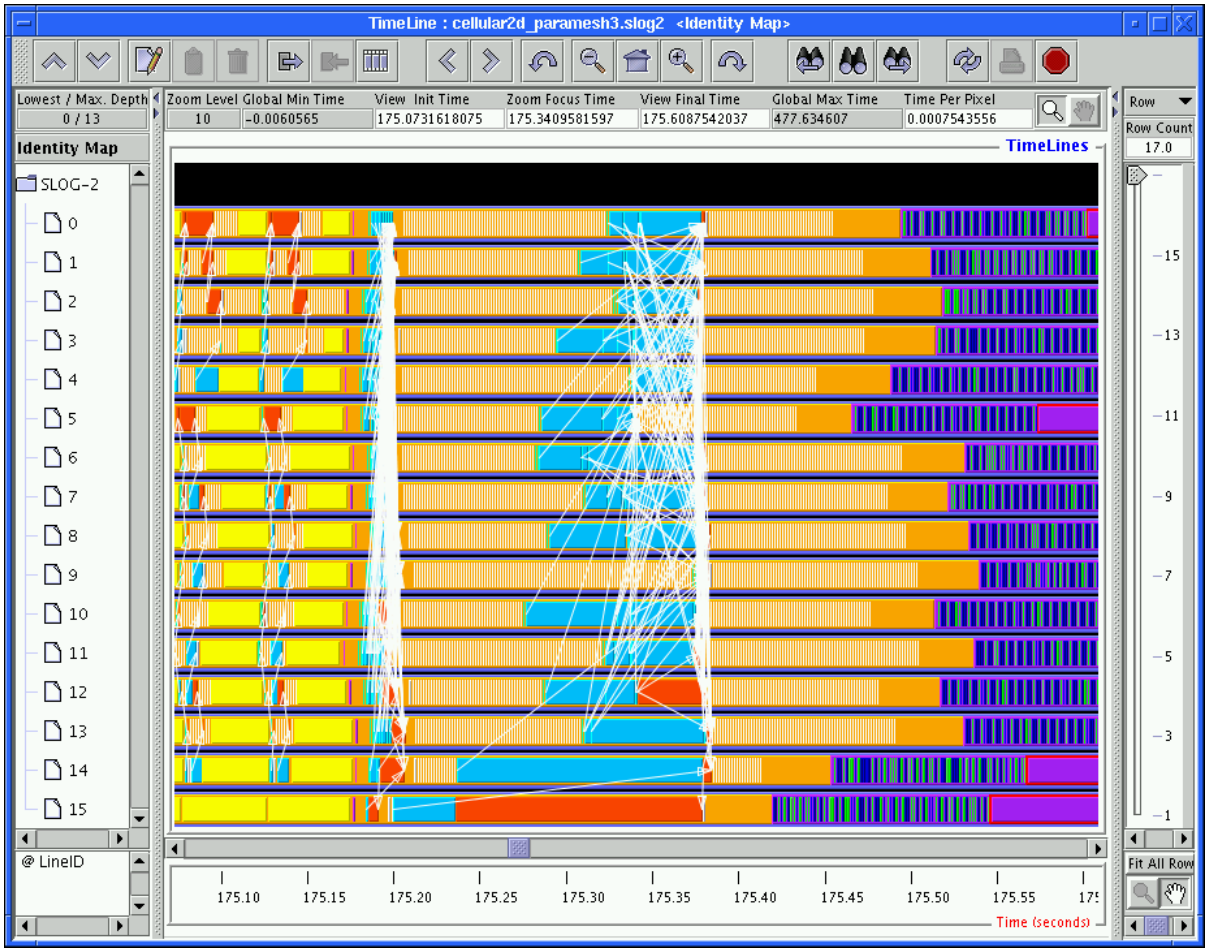

Figure 2.5: Performance analysis of MPI-based communications using Jumpshot [31]

Finally, there has been some work on performance monitors that use software drivers to monitor FPGA-based applications. Airwolf [32] is an example of FPGA-based profiling tools for soft-processor applications. Airwolf is designed for Intel's NIOS-II processor. It measures the time spent in segments of applications by adding software drivers to the beginning and end of code segments. Similar to most software monitors, Airwolf adds an overhead cost due to using software drivers.

2.4 Hardware-based Performance Monitoring

As mentioned in Section 2.3, most software-based performance monitors collect performance data by adding instrumentation code to programs or by using software drivers. On the other hand, performance analysis on a hardware level requires access to registers and counters in the lower levels of machines. For reconfigurable hardware, such as FPGAs, the hardware must be added to allow performance analysis. For non-reconfigurable devices, such as CPUs and GPUs, devices must be manufactured with performance analysis tools (such as counters and registers) already in place. Then special software must be developed to provide access to the hardware-level tools. From the monitors mentioned in Section 2.3, VTune and CodeXL have such characteristics and can also be categorized as hardware-based monitors. In this section, we discuss some of the hardware monitoring techniques as well as related work on hardware-based performance monitors.

Similar to manual software-level performance analysis, there are hardware-level tools for manual debug and analysis of digital design. Quartus SignalTap [33] and Vivado Integrated Logic Analyzer (ILA) [34] are examples of manual debugging tools for Intel and Xilinx FPGAs, respectively. These tools allow runtime system-level debugging of internal FPGA signals by triggering on user-defined conditions. In addition to functional debugging, logic analyzers can occasionally be used for simple performance debugging. Event sampling in logic analyzers is typically done using circular buffers (Figure 2.6), where data is continuously sampled until the trigger condition is met[2]. Circular buffers are especially useful when it comes to buffering data streams. These buffers work as if they are connected end-to-end; hence, when the buffer is full, the new data is written over the oldest data. The user usually has the freedom to place the trigger condition in any position within the bounds of the buffer. Therefore, the user can choose how much of the buffered data are the events that happen prior to the trigger condition and how much of the buffered data are the subsequent events to the trigger condition.

Although logic analyzers are powerful tools for functional debugging, there are many limitations when it comes to performance debugging. First of all, logic analyzers are meant for the manual debug of FPGAs. This is typically undesirable when it comes to performance analysis and adds constant manual work or calculation that can be easily done automatically. Secondly, the size of sampling buffers is limited by the amount of on-chip memory that is not used by the circuit that is being debugged. This hugely constrains tasks such as event tracing to events that are happening in a short span of time. Finally, logic analyzers are meant for single-FPGA debugging and are unable to perform multi-FPGA analysis.

AXI Performance Monitor [35] is a Xilinx IP that allows real-time performance monitoring of different AXI interfaces (such as AXI-Stream and AXI-Lite). The AXI Performance Monitor is capable of logging certain AXI events and providing a timestamp difference between two successive events. It is also capable of counting the number of AXI events. In addition, the AXI Performance Monitor allows run-

[2]Data sampling may continue further depending on the choice of trigger position

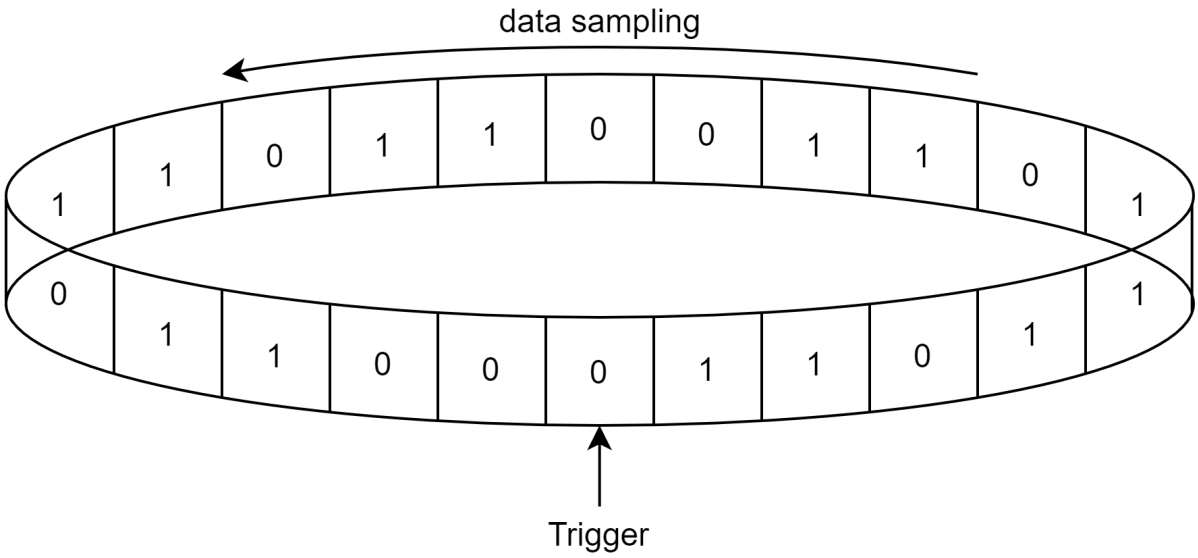

Figure 2.6: Data sampling using circular buffers

time measurements of applications and also provides throughput measurements of AXI lines. The AXI Performance Monitor is a very useful tool for performance monitoring of single FPGAs. However, it does not suit the needs of multi-FPGA systems. The timestamps provided by this core are local to the FPGA it is used in; therefore, they cannot be used to determine the order of events in a datacenter. In addition, the AXI Performance Monitor is limited to AXI line width of 32 bits and does not support AXI lines with larger widths.

2.4.1 Related Work

PAPI (Performance Application Programming Interface) [36] is a set of performance monitoring tools for both software and hardware related events. It provides access to an interface to monitor hardware performance counters on microprocessors. PAPI is intended to help high-level software designers to monitor the performance of their software program using real-time hardware monitors.

SnoopP [37] and SoCLog [38] are examples of performance analysis tools meant for single-FPGA monitoring. SnoopP is a non-intrusive FPGA-based performance monitor for hardware/software co-design. It provides cycle-accurate performance data of a software running on a soft processor instantiated on an FPGA. SnoopP uses a number of counters and comparators to determine how long the Program Counter value falls within a certain range. Using this technique, it can determine the execution time of certain sections of an application. The goal of SnoopP is to provide designers with a quick analysis of which parts of the design are too slow to be implemented in software given critical timing constraints. This can help designers make faster partitioning decisions.

SoCLog is an FPGA-based performance monitoring framework designed in High-Level Synthesis (HLS). It collects traffic data by snooping on AXI4-Stream bus transactions. SoCLog collects timestamps of the start and end of certain events and uses a software GUI to visualize them in a timeline. This helps designers understand which parts of a program take the most time and where performance bottlenecks are.

The work by Nunes et al. [39] describes a profiler for a heterogeneous multi-core multi-FPGA cluster [40]. This profiler uses a set of counters and tracers that are placed within the FPGAs for hardware-level profiling. The profiled data is stored using on-chip memory in the FPGA and later sent to remote workstations for analysis. The inter-FPGA communication is done using the MPI Parallel Environment (MPE) [28]. Software-level profiling is done by adding instrumentation code into MPI functions. Finally, JumpShot is used to visualize the profiler logs.

Moongen [41] is a performance monitor that uses scriptable packet generators. It deploys Precision Time Protocol (PTP) for time synchronization. Using commodity NICs for hardware timestamping, Moongen can perform latency measurements with sub-microsecond accuracy. It also uses a packet-reception API to collect throughput data. Moongen uses the Data Plane Development Kit (DPDK) [42] framework and is restricted to hardware that supports DPDK features.

2.5 A Discussion on the Related Work

Sections 2.3 and 2.4 discussed a wide variety of performance monitors and profilers. Out of all the discussed tools, few meet all the criteria required to monitor a typical multi-FPGA system. In this section we present a brief comparison between the tools discussed in Sections 2.3 and 2.4. Finally, we discuss how Pharos compares to the existing tools in terms of capabilities, what ideas it borrows from the existing solutions, and what it offers differently.

Table 2.2 shows a comparison between some of the related work that were discussed in Sections 2.3 and 2.4. The monitors are categorized into four categories of single-CPU, multi-CPU, single-FPGA, and multi-FPGA monitors. For each monitor, the measurement and analysis types are shown in this table. The runtime measurement represents the capability to measure the amount of time an application or a segment of an application takes to run. The latency measurement is the ability to measure point-to-point latency between network nodes, therefore, it does not apply to single-CPU and single-FPGA monitors. The throughput measurement is measuring the amount of data transmitted in a data bus over a certain amount of time.

The type of analyses provided by these monitors include event logs, memory analysis, and single- and multi-thread analysis. Event logs help determines the order of events within the system. In this table, a wide variety of capabilities from event tracing in multi-machine systems to call graph analysis of single-CPU routines are categorized as event logging. Finally, memory and threading analyses refer to monitoring memory access of applications and collecting data about applications that use single- or multi- thread computing, respectively. The latter only applies to CPUs.

All of the software-based monitors are to some degree intrusive. These monitors either add code segments to applications or use software drivers, which generally leads to higher overhead latency values in comparison to hardware-based monitors. Among the existing FPGA-based monitors, the Xilinx AXI Performance Monitor (APM) and the SoCLog offer event logging capabilities very similar to what Pharos aims to offer. However, the measurements offered by the APM and the SoCLog are meant for single-FPGA analysis and do not support multi-FPGA systems. On the other hand, Moongen offers many multi-FPGA monitoring capabilities. However, these measurements are limited to hardware that support DPDK applications.

To the best of our knowledge, none of the existing solutions offer latency and traffic measurements as well as event logging capabilities for multi-FPGA systems and also remain generic enough to be used in

Table 2.2: A comparison between some of the related work and Pharos

Monitor/ Profiler	Monitor Type	Target Platform	Measurement Types			Analysis Types		
			runtime	latency	throughput	event logs	memory	threading
GProf [23]	single-CPU	Unix Apps	×			×		
VTune [26]	single-CPU	Intel CPUs	×			×	×	×
Datadog [29]	multi-CPU	Generic	×	×	×	×	×	×
MPE [28]	multi-CPU	MPI programs	×	×		×		
Xilinx APM [35]	single-FPGA	Xilinx FPGAs	×		×	×		
SoCLog [38]	single-FPGA	Generic	×		×	×		
SnoopP [38]	single-FPGA	Generic	×					
Moongen [41]	multi-FPGA	DPDK programs	×	×	×			
Nunes et al. [39]	multi-FPGA	TMD servers [40]	×		×	×		
Pharos	multi-FPGA	Generic	×	×	×	×		

a wide variety of applications. Pharos, the performance monitor presented in this work, aims to achieve such target. Pharos borrows some traffic monitoring techniques, such as throughput measurements and event logging, from the Xilinx AXI Performance Monitor and SoCLog and expands them to multi-FPGA systems. In addition, Pharos also offers unidirectional latency measurements between FPGAs. The goal of Pharos is to remain as generic as possible. This means that Pharos does not depend on a specific type of datacenter or a specific type of network protocol to monitor the performance of multi-FPGA systems.

Chapter 3

The Pharos Performance Monitor

This chapter discusses the architecture details of Pharos and its components. Pharos is a collection of open-source [8] hardware modules, written mostly in C++ for Vivado High-Level Synthesis [43] and SystemVerilog. An overview of the Pharos architecture is shown in Figure 3.1. The overall architecture of Pharos can be divided into four main blocks: a timer, a PTP block, a Latency Monitor, and a Traffic Monitor. All of these blocks and their sub-blocks are standalone cores that can be added or removed independently based on the user's needs. All the external communications of Pharos (either between two instances of Pharos or between Pharos and other FPGA applications) are done through AXI-streams. This makes Pharos independent of the underlying network protocols allowing the users to swap the lower communication layer without affecting the functionality of the performance monitor. The Latency Monitor performs unidirectional point-to-point latency measurements, while the Traffic Monitor provides information such as throughput and bandwidth by snooping on AXI lines. The Traffic Monitor also includes an event logger that logs certain events and provides timestamps whenever they occur. The rest of this chapter discusses the hardware and the tools used for the implementation of Pharos and the use of time synchronization in the performance monitor. The details of the time synchronization protocol are discussed in Section 3.3. The Latency Monitor and the Traffic Monitor are discussed in Sections 3.4 and 3.5, respectively.

3.1 Implementation

All of the blocks and sub-blocks of Pharos are meant for FPGA implementation. We implemented Pharos on Xilinx Ultrascale+ XCZU19EG-FFVC1760-2-I FPGAs [44], which are used on Fidus Sidewinder boards. In addition to the FPGA, an ARM CPU is included on the same silicon die. The communication between the FPGA and the ARM CPU is through several high-speed AXI channels. The FPGA is part of the ZU19EG series [45], which is a mid-range series of FPGAs with 1.1 million logic units and 9.8 MB of on-chip memory. The Xilinx CAD tools were used in the development of Pharos. Particularly, Vivado HLS [43] was used for High-Level Synthesis development, and Vivado 2018.1 [20] was used for the development of Verilog/SystemVerilog IPs and the integration of the components of Pharos. For our implementation, we used a 100 MHz clock for all blocks within Pharos.

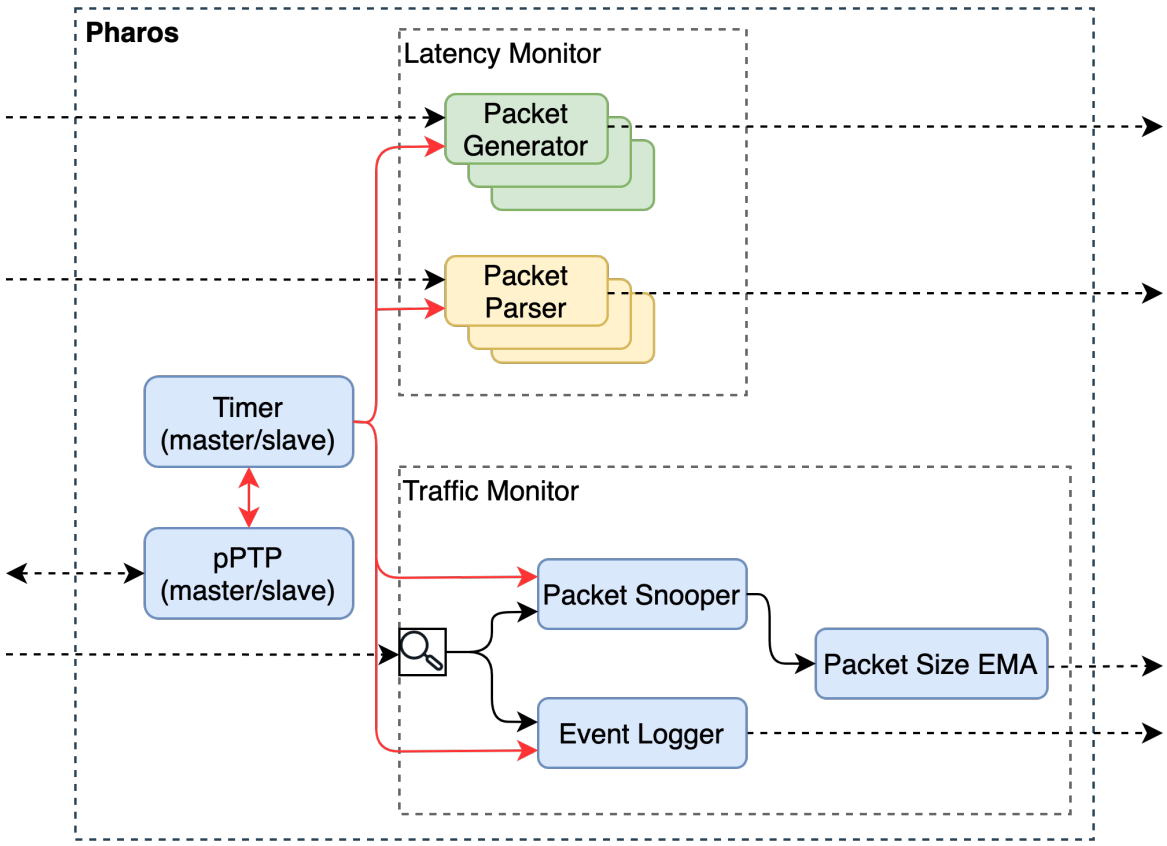

Figure 3.1: High-level architecture of Pharos

3.2 The Idea of Global Time

Performance monitoring of events within one FPGA is normally a fairly simple task. First, modules within an FPGA can be controlled using the same signals. For instance, one signal can be used to start or stop several different tasks. This makes it easier for the monitor to orchestrate many measurements simultaneously. Secondly, hardware modules within an FPGA ordinarily use the same clock signal. Therefore, one timer can be used to measure the duration of tasks within these modules. Additionally, external factors such as network traffic do not affect the performance of tasks within an FPGA.

On the other hand, performance monitoring is more challenging when it comes to events that are distributed over many FPGAs. To conduct accurate measurements in multi-FPGA systems, the monitor must establish a common point of reference for all FPGA devices. In Pharos, this reference point is established using the idea of *global time*. Global time is a time that all FPGA devices agree on, even though they use different clock sources. The main objective of using a global time is to monitor tasks as if they are running on the same FPGA. This helps the monitor abstract away the network that connects the FPGAs and provides the means to determine the sequence of events that are happening in separate FPGAs across the network.

The idea of global time has several benefits. Firstly, when all FPGAs agree on a common time, the need for round-trip latency measurements is eliminated and we can conduct unidirectional latency measurements between different nodes of a datacenter. Unidirectional measurements are both faster than round-trip measurements (twice as fast if the network traffic is symmetric in a round-trip measurement)

and cause less traffic since the packets do not have to be sent back to the packet source as they are in round-trip latency measurements. Additionally, using a global time provides the means to determine the sequence in which certain events are happening in various nodes within a datacenter. This allows the monitor to trace events in a multi-FPGA system.

It is important to note that the time synchronization between FPGAs must happen regularly. The reason for this lies within the physical properties of the clock generation crystals used on the FPGA boards. While the crystals of two different boards may be designed to generate the same clock frequency, small variations in their physical properties will cause the frequencies of the FPGAs to be slightly different. Over time, these small frequency variations accumulate and cause the times of nodes to drift apart from each other. Pharos uses a modified version of the Precision Time Protocol (PTP) to synchronize times between FPGAs. The timer is a 64-bit unsigned counter that starts when the FPGA is programmed. Using a 100 MHz clock, it can run for over 5800 years before overflowing. The timer and the PTP blocks must be either a master or a slave instance. To ensure the integrity of the global time, only one instance of Pharos must contain the reference timer (the only PTP-master in flat PTP networks and the grand-master timer in hierarchical networks). All other instances will synchronize their times to the reference timer using PTP communications.

3.3 Time Synchronization in Pharos

The decision of choosing a time synchronization protocol was a crucial one. As explained in Section 3.2, the idea of global time plays a critical role in both the functionality and the efficiency of Pharos. To make this decision, we looked at two of the most used synchronization protocols: NTP and PTP, previously described in Sections 2.1.1 and 2.1.2, respectively. While NTP is generally easier to implement, it is more suitable for application-level synchronization. NTP offers synchronization in the range of milliseconds, which is slower than the usual accuracy requirements of today's FPGA performance debugging. Modern FPGA applications run at clocks in the range of tens or hundreds of megahertz. With a millisecond accuracy, the performance monitor can at best reach a granularity of tens of thousands of cycles. On the other hand, PTP provides a sub-microsecond accuracy and can provide cycle-accurate synchronization, which is much more suitable for monitoring FPGA applications. Taking advantage of the programmability of FPGAs, this work introduces the modifications made to the PTP standard so it better suits the needs of multi-FPGA systems. In the rest of this thesis, "pPTP" refers to "Pharos PTP", the modified version of the protocol used in this work. The rest of this section discusses the details of pPTP.

3.3.1 pPTP - Modifications to PTP

The protocol described in Section 2.1.2 is an implementation for commodity hardware. To provide time-synchronization support for a network, either the network switches and routers must be PTP-aware or such hardware must be purchased separately and put in the Ethernet PHY of network nodes, in which case it is the responsibility of the datacenter administrator to ensure these devices are compatible with the rest of the network. On the other hand, FPGAs not only remove the requirement of using commodity hardware, but also provide the flexibility to modify the PTP protocol to better suit the needs of each datacenter. This section discusses the modifications made to the PTP protocol and the architecture of pPTP.

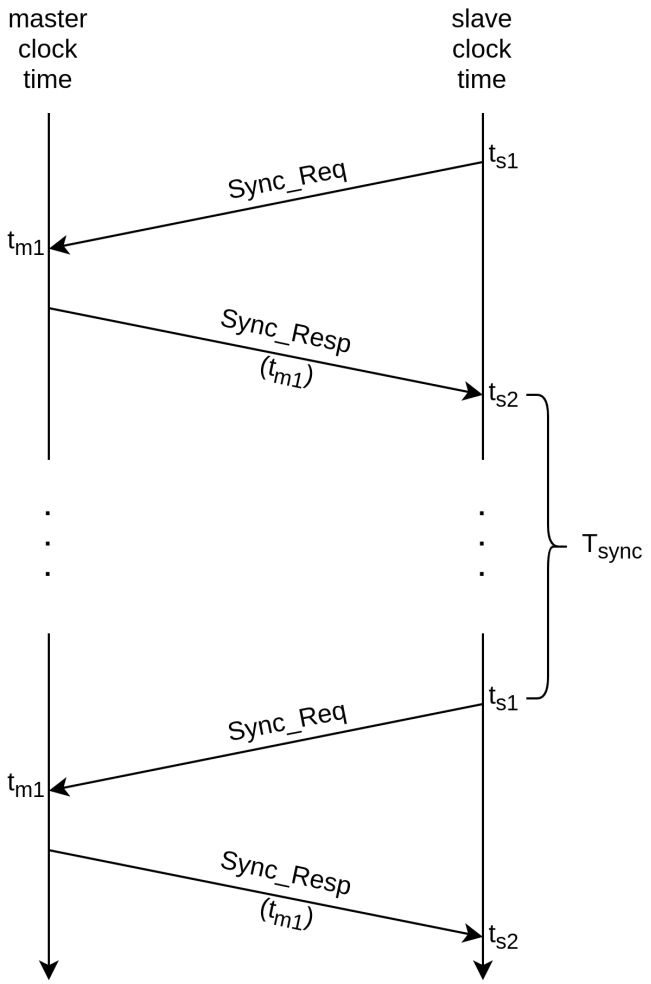

Figure 3.2: Message sequence of pPTP

Figure 3.2 shows the message sequence of pPTP. The initial *sync* message of PTP is eliminated in pPTP. Therefore, it is up to the PTP-slave to initiate a time synchronization. The new message sequence is as follows:

1. Slave sends a synchronization request (*Sync_Req*) and records its own time (t_{s1})

2. Upon receiving the *Sync_Req* message, the master sends a response back with a timestamp of the master's current time (t_{m1})

3. Upon receiving the synchronization response (*Sync_Resp*), the slave records its own time (t_{s2})

By the end of one synchronization sequence, three timestamps are known by the slave: (t_{s1}), (t_{s2}), and (t_{m1}). Using these timestamps, the slave can calculate the network propagation delay (Equation 3.1).

$$t_{network} = \frac{(t_{s2} - t_{s1})}{2} \tag{3.1}$$

This value represents a unidirectional trip of a PTP packet between slave and master instances. Note that this equation still holds the assumption of symmetrical propagation delays, explained in Section 2.1.2. Equation 3.2 is used to calculate the new time. The slave will then update its time from t_{s2} to t_{s_new}.

$$t_{s\_new} = t_{m1} + t_{network} \tag{3.2}$$

A time-out parameter is defined for each slave instance, which can be changed prior to synthesis. If a *Sync_Resp* is not received within the time-out interval, a new *Sync_Req* message will be sent to the master. If *Sync_Resp* is received, Steps 1 to 3 will be repeated after a synchronization interval, T_{sync}. Pharos defines a separate T_{sync} value for each PTP-slave instance. This means each slave can synchronize its time to the master-timer at its own rate, which can be adjusted based on how quickly the slave time drifts apart from the master time.

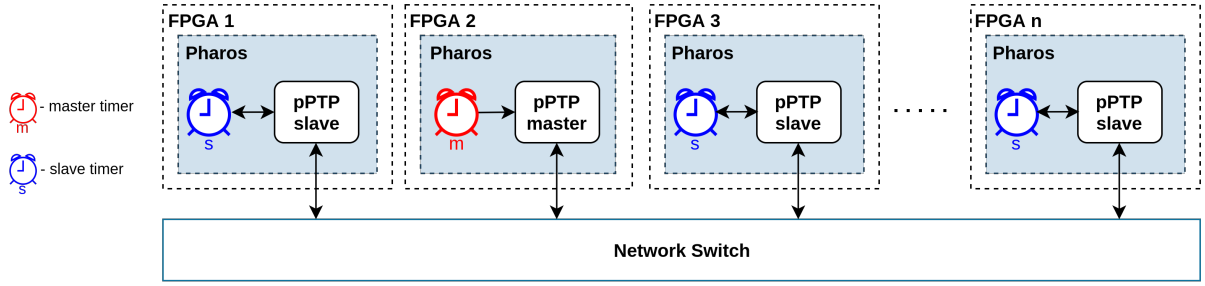

Figure 3.3: A flat pPTP network. The master node is chosen arbitrarily.

Since it is up to the slave instances to initiate a synchronization procedure, there is no need for the PTP-master instance to either have knowledge of the slaves or broadcast its time to the network. However, Slave instances must know the network address of the PTP master. This provides Pharos with the freedom to connect and disconnect slaves on the fly. The maximum number of slaves that can connect to the pPTP master is limited by the free space left on the AXI-interface address space, which is defined by the width of the TDEST channel. The current version of Pharos uses a 16-bit TDEST signal, which can support up to 65,536 unique addresses. Since this address space is shared by all hardware modules that use the AXI-interface, normally only a portion of it can be used by pPTP. However, the size of the TDEST channel can easily be changed to allow more devices to be connected to the pPTP network. Appendix B provides further details on the interface and the usage of pPTP.

3.3.2 pPTP Hierarchy

Figure 3.3 shows a pPTP network. We call this a *flat* network since it only contains one level of hierarchy. This means that there is only one master clock within the whole system and every other instance of pPTP must be a slave instance. There are no real differences between the master and slave timers. Both timers are of the same size and they both increment time every cycle. The only difference is that the time of the slave timers can be updated by pPTP slaves. The direction of arrows in Figure 3.3 is meant to show the direction of messages (in this case, it shows that pPTP-master instances only read time from master timers, but pPTP-slave instances both read time and update the time of slave timers). In the current version of Pharos, there are no algorithms to automatically choose the master instance

and it must be chosen manually. Ideally, the master clock must be the clock with the most stable clock generation crystal. In Section 4.2, we characterize the accuracy of pPTP instances. This work can be used in the future as a master clock selection apparatus.

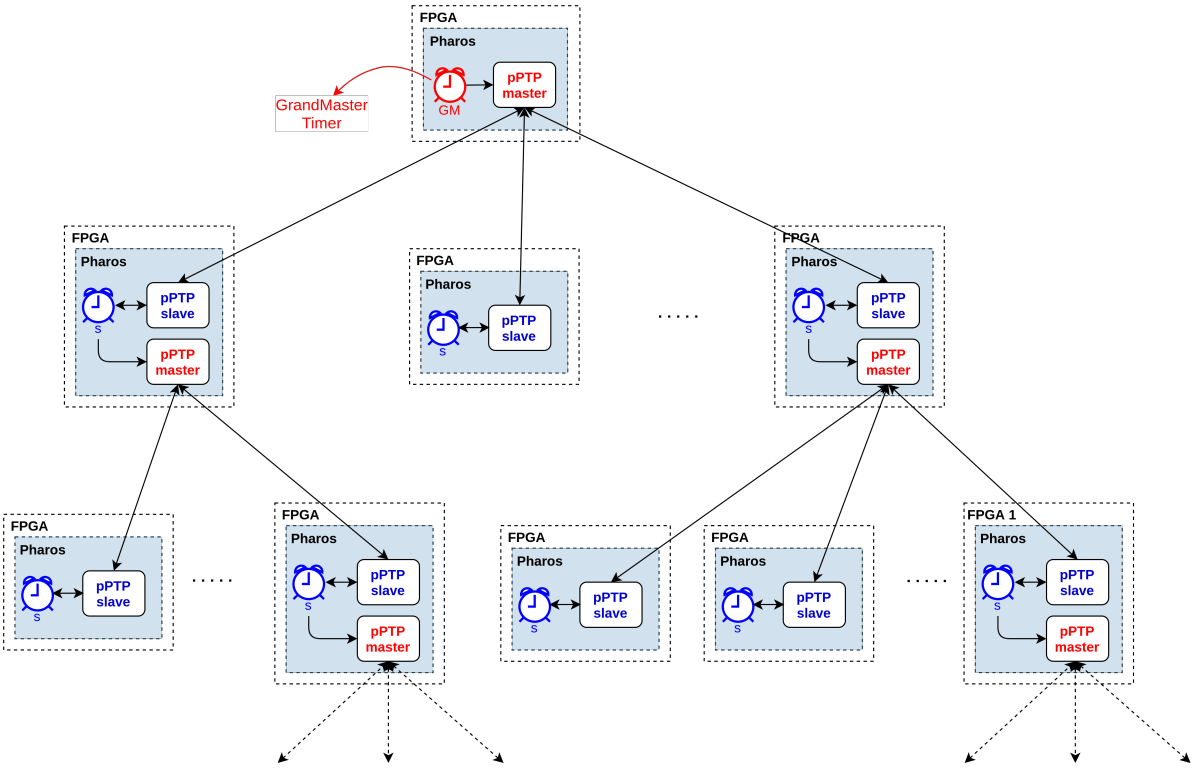

Figure 3.4: Hierarchical structure of pPTP

Flat pPTP networks will work for any network size as long as each slave can be assigned a unique address using the TDEST signal. Increasing the number of slaves does not affect the FPGA resource utilization of either the master or the slave instances (resource utilization is discussed in Section 4.5). However, connecting a large number of slaves to one master may cause congestion on the network node that is holding the master instance. This is because the master instance has to respond to the requests from all of the slave instances in the pPTP network. To reduce network traffic on the master instance, pPTP can be arranged in a hierarchical structure. The main goal of the hierarchical structure is to distribute the network traffic among several sub-trees and avoid flowing all of the traffic towards one node. Figure 3.4 shows such structure. In a hierarchical structure, one node is chosen to hold the *GrandMaster Timer*, which provides the reference time for the whole network. In the hierarchical structure, some nodes must contain both pPTP master and pPTP slave instances. In these nodes, the slave instance of pPTP synchronizes to the pPTP master of its parent node and updates the timer in its own node. Then the pPTP master of the same node uses the same timer to provide timestamped packets for the nodes in its sub-tree.

3.4 The Latency Monitor

The Latency Monitor is a set of hardware modules that enables point-to-point latency measurements between two instances of Pharos. This can be either the latency between two points within the same FPGA or between nodes that reside in two different FPGAs. Pharos itself is unaware of where the nodes are located and the latency measurement is independent of the whereabouts of the monitors. The Latency Monitor consists of two main components: the *Packet Generator* and the *Packet Parser*. Figure 3.5 shows the architecture of the Latency Monitor. Each instance of the Latency Monitor can contain any arbitrary number of Packet Generators and Packet Parsers. This number must be determined before synthesis. Performing unidirectional latency measurement between two nodes requires both nodes to agree on a common time. Therefore, Latency Monitors rely on time synchronization using pPTP to conduct reliable unidirectional measurements. The timer symbol in Figure 3.5 indicates that the Packet Generators and the Packet Parsers are time-aware (they take time as an input). The timer can be a master timer or a slave timer, depending on what pPTP instance the FPGA holds. The rest of this section explains the details of the Latency Monitor.

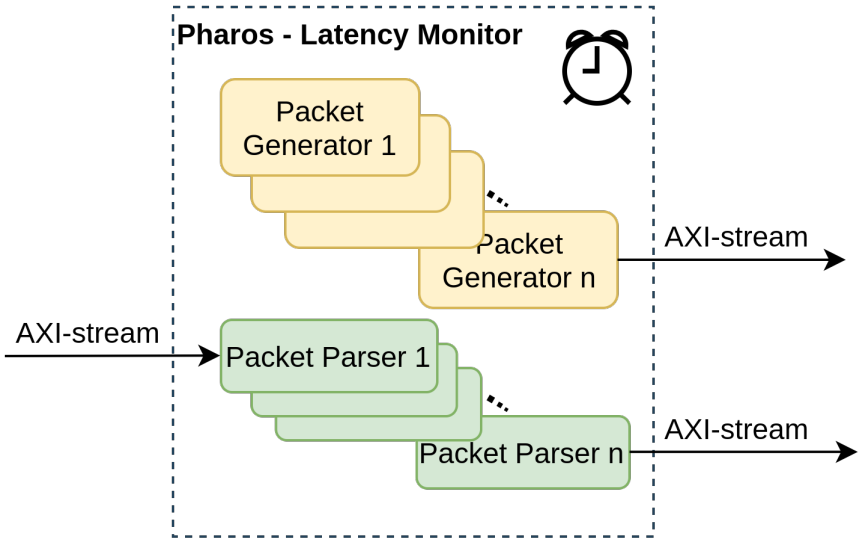

Figure 3.5: High-level architecture of the Latency Monitor

3.4.1 Packet Generation

Packet Generators generate packets that are exclusively used for latency measurements. These packets are timestamped and sent to a Packet Parser of the destination. The destination, which is embedded into the TDEST field of the AXI-Stream, is determined by the user and can be changed on the fly. As mentioned earlier, for a reliable unidirectional latency measurement, the time of the two ends of the measurement must be synchronized using pPTP. Each Packet Generator can be connected to only one Packet Parser (each Packet Parser, however, can connect to many Packet Generators, this is discussed in Section 3.4.2). Packet Generation happens at a certain frequency that is determined by the user and can be changed at run-time. Furthermore, packet generation can be started or stopped at any time using an input register. The I/O ports of the Packet Generator are summarized in Table 3.1.

Table 3.1: Packet Generator Port List

Port Name	Direction	Description
clk	input	clock signal
aresetn	input	asynchronous active low reset signal
time	input	the current time of the node
init	input	initializes packet generation
gen_frequency	input	frequency of packet generation
out_stream	output	AXI-Stream - output stream of timestamped packets

3.4.2 Packet Parsing

The main goal of the Packet Parser is to read the timestamped packets and measure the point-to-point latency. This is done using two pieces of information: the time that the packet was generated (which is encapsulated in the timestamped packets from the Packet Generators) and the current time of the node that the Packet Parser is in. Similar to the Packet Generators, the time that the Packet Parser reads must be synchronized to the network time using pPTP to ensure reliable latency measurements. Each Packet Parser can receive packets from several Packet Generators. The maximum number of Packet Generators is determined by a parameter prior to synthesis. Packet Generators may connect/disconnect on the fly as long as the total number of connections does not exceed the maximum limit. Finally, it is possible to route the input stream of Packet Parsers to their output. This can be used to daisy chain many Latency Monitors and conduct measurements at each point (see Section 3.4.3). Table 3.2 summarizes the ports of Packet Parser.

Table 3.2: Packet Parser Port List

Port Name	Direction	Description
clk	input	clock signal
aresetn	input	asynchronous active low reset signal
in_stream	input	AXI-Stream - input stream of timestamped packets
time	input	the current time of the node
measure	input	input register used to start/stop the latency measurements
output_frequency	input	frequency for outputting latency values
smooth_factor	input	smoothing factor of the exponential moving average
out_stream	output	AXI-Stream - output stream of timestamped packets
output	output	AXI-Stream - the current latency measurement

Multi-Fan-In Connections

As previously mentioned, each Packet Parser can connect to many Packet Generators simultaneously. Handling a dynamic number of connections on the fly is not a trivial task. To implement multi-fan-in support for the packet Parsers, two blocks of memory are used. This can be done in Vivado HLS using two arrays, which synthesize into block RAMs. The arrays are of the same length, which is equal to the maximum number of connections. This must be determined by the user prior to synthesis. The Packet Parser keeps track of the number of Packet Generators that it is connected to (referred to as *active elements*). This number gets updated every time a new Packet Generator gets connected or disconnected.

Listing 3.1 shows the declaration of the aforementioned memory blocks. The *connections* array stores the TDEST of connected Packet Generators. This is useful to indicate which Packet Generator the latency belongs to when latency values are read out. The second, *elements*, keeps track of information about each connected node. Each node of this array is of type *connected_node*, which is a struct that stores specific information about each Packet Generator. This information includes the average latency between the Packet Generator and this Packet Parser and whether or not the Packet Generator is connected (whether or not it is an *active element*). The information about each Packet Generator is stored in a cell with an index equal to the TDEST of that Packet Generator.

```
struct connected_node {
        ap_uint <1> connected = 0;
        ap_f latency_EMA = 0;
        ap_f latency_EMA_prev = 0;
        ap_uint <64> latency = 0;
};

//initialize number of connected packet generators to 0
static ap_uint<16> active_elements = 0;

// declare memory to hold the indices of the connected elements
static ap_uint<64> connections [MAX_NUM_OF_ELEMENTS];

// declare memory to contain data for each element.
static connected_node elements [MAX_NUM_OF_ELEMENTS];
```

Listing 3.1: Packet Parser multi-fan-in connection handling. Packet Parser stores the information of each connection by the declaration of memory arrays in Vivado HLS

Output Generation

Packet Parser is able to output latency every cycle, but this is an expensive task in terms of FPGA resources. This is because the rate at which data is being read out of the FPGA is much slower than the rate of data Generation. Therefore, the data must be stored in on-chip memory of the FPGA, which is a costly task. To rectify this, we limit the frequency of Packet Parser outputs by continuously calculating an Exponential Moving Average (EMA) of the latency values for each connected node. The EMA is calculated by Equation 3.3,

$$EMA_i = (1 - \beta)EMA_{i-1} + \beta(L_i) \tag{3.3}$$

where EMA_i is the current moving average, EMA_{i-1} is the previous moving average, L_i is the current latency value, and β is a smoothing factor between 0 and 1. As β gets closer to 0, the moving average takes more history into account and the curve becomes smoother. β is taken from the user as an input. The average is updated every packet, but it can be read out at a lower frequency. Each Packet Parser takes in a parameter that determines the output interval. This parameter, which can be changed

at run-time, indicates how often the user likes to read out the latency numbers. Whenever this interval is reached, the Packet Parser outputs the address (TDEST) of each connected Packet Generator followed by their respective latency EMA values.

3.4.3 Measuring Latency

Figure 3.6 shows a typical unidirectional latency measurement procedure between two instances of Pharos. If the two instances of Pharos are on different FPGAs, then they must synchronize their times using pPTP. If the nodes are in the same FPGA, there is no need for pPTP synchronization since they have access to the same timer. Both the Packet Generator and the Packet Parser use AXI-Stream for external communications. However, communication between instances of Pharos may be done using other protocols. This is only relevant if the two Pharos instances reside in different FPGAs, in which case AXI-Stream packets can be encapsulated within packets of other protocols (such as UDP, TCP, or ethernet) to be sent over the network.

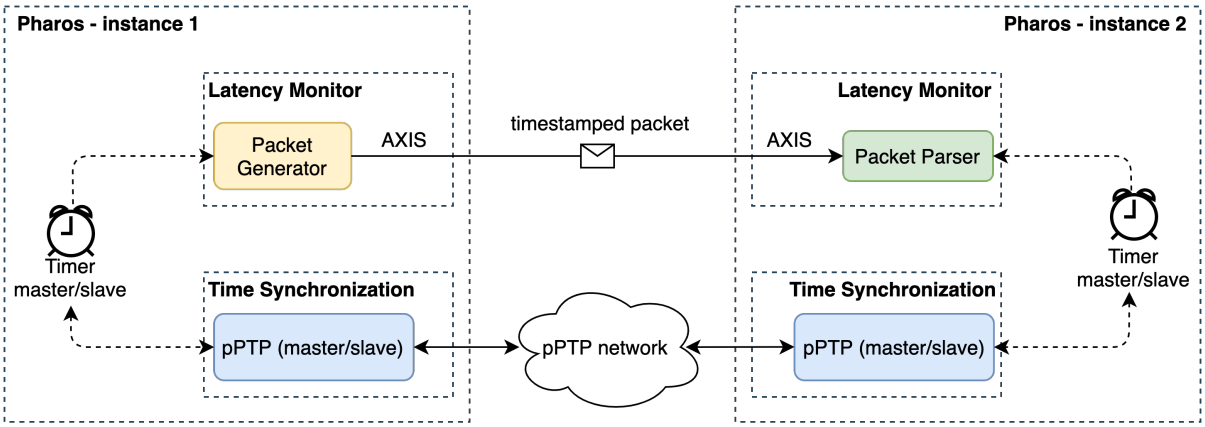

Figure 3.6: Unidirectional latency measurement using the Latency Monitor

As mentioned previously, each Packet Generator can send packets to only one Packet Parser. However, the Packet Parsers are multi-fan-in and can connect to multiple Packet Generators simultaneously. The destination address of packets is determined using the TDEST field of the AXI interface. In the current version of Pharos, this field is set to 16 bits. Therefore, the Packet Parser can calculate latency between its node and 65536 other nodes at the same time. It is worth noting that supporting many nodes at the same time can be expensive in terms of FPGA resources. To avoid using more resources than necessary, the maximum number of connections for a Packet Parser can be set through a parameter before synthesis. This parameter only indicates the *maximum* number of Packet Generators that can connect to a Packet Parser, though the actual number of connections can be smaller and can change on the fly. This means that a Packet Generator can connect to or disconnect from a Packet Parser at run-time as long as the number of connections to that Packet Parser does not exceed the maximum limit.

As mentioned in Section 3.4.2, Packet Parsers can output every timestamped packet they receive. This makes it possible to arrange Latency Monitors in a daisy chain, similar to Figure 3.7. It is important to note that Packet Parsers do not generate any new timestamps and only forward their input streams. Therefore, the latency measured at each stage is the latency from the Packet Generator (the very first

node of the chain) to that Packet Parser. For example, the latency value L_2 represents the latency from Latency Monitor 1 to Latency Monitor 3. Since Packet Parsers are pass-through hardware, each stage adds exactly one cycle to the latency.

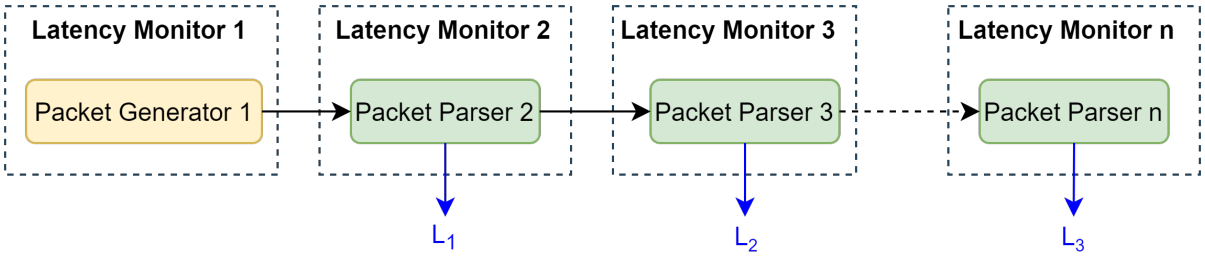

Figure 3.7: Daisy chaining many Latency Monitors

3.5 The Traffic Monitor

The idea of the Traffic Monitor is to observe the traffic within the FPGA and log certain information. The main functionality of the Traffic Monitor is to count the number of passing flits and packets. Knowing the amount of traffic and the duration of measurement, Pharos can then calculate throughput. The Traffic Monitor is also able to log certain events and provide timestamps whenever they occur. Using the idea of global time in Pharos, event logging can be used to trace events over the entire network of FPGAs. Figure 3.8 shows the high-level architecture of the Traffic Monitor. The monitor uses the AXI interface to look at the traffic; however, no traffic passes through the monitor. Instead, the passing traffic is merely observed. This means that the monitor does not output a TREADY signal and cannot back-pressure on the AXI-stream line. Section 3.5.1 discusses the interface of the Traffic Monitor. The sub-components of the Traffic Monitor are discussed in Sections 3.5.2 and 3.5.3.

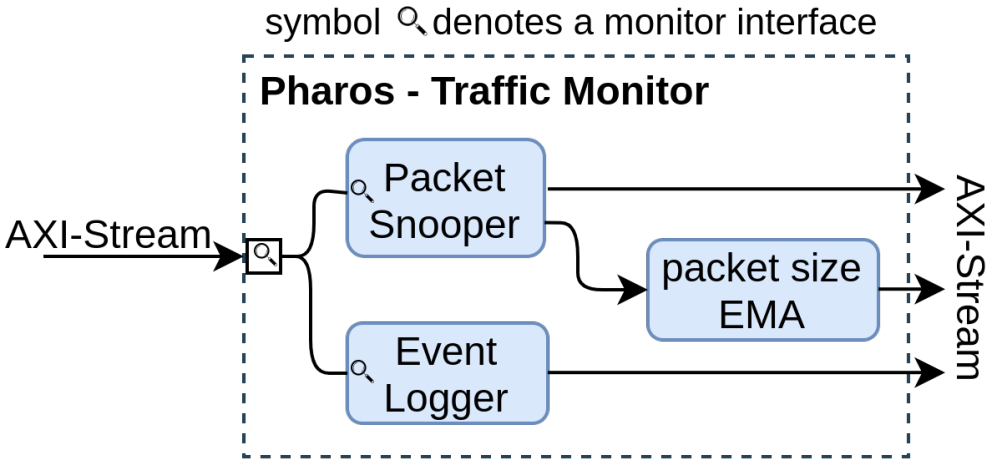

Figure 3.8: High-level architecture of the Traffic Monitor

3.5.1 Monitor Interface

The Traffic Monitor snoops on AXI-Stream lines. In particular, the Traffic Monitor looks at five of the AXI-Stream channels: TVALID, TREADY, TKEEP, TLAST, and TDATA. The Traffic Monitor is not pass-through hardware, meaning that none of the AXI traffic passes through the monitor. Therefore, the Traffic monitor cannot back-pressure on any traffic. This is done by making the input stream a *Monitor Interface* in *Vivado IP Packager*. Figure 3.9 shows a simplified view of how the Traffic Monitor interacts with the passing traffic. In this case, the Traffic Source snoops on the traffic that is going from the Traffic Source to the Traffic Sink. In contrast to the usual AXI-Stream transactions, the Traffic monitor does not output a TREADY signal and takes TREADY as an input. An AND product of the TREADY and TVALID signals is used to indicate a valid transaction between the Traffic Source and the Traffic Sink.

3.5.2 Packet Snooper

Packet Snooper is the main component of the Traffic Monitor. It snoops on an AXI-Stream line and collects information about the passing traffic. Packet Snooper has a measurement enable signal that can be written to by the user. When this enable signal is high, the Packet snooper starts collecting data. This data is sent to the output when the negative edge of the measurement enable is detected. During the measurement, the following data is collected by the Packet Snooper:

- number of passing flits
- number of passing packets
- size of each packet
- duration of the measurement

As previously mentioned, the Packet Snooper does not output a ready signal. It uses an AND product of the input TREADY and TVALID signals to detect a valid transaction. Each transaction indicates a passing flit of data. To count the number of packets, the Packet Snooper looks at the TLAST signal of the AXI interface. A high TLAST signal indicates the end of a packet. The combination of the number of packets and the duration of measurement provides a way to compute throughput.

To measure the size of packets, the Packet Snooper calculates the size of each flit using the TKEEP signal of the AXI-Stream. The TKEEP signal indicates how many bytes of each flit is part of the data stream. The width of the TKEEP channel is always one-eighths of the length of the TDATA channel since each bit of TKEEP is a qualifier for a byte of the payload. The size of each flit is calculated based on how many byte qualifier bits are asserted for each transaction. The Packet Snooper keeps summing up the flit sizes until it detects the end of the packet. To avoid reading out the size of every packet out of the FPGA, the Packet Snooper feeds the packet size data into the Packet Size EMA (See Section 3.5.3), which averages the packet size data over longer periods.

3.5.3 Packet Size Averaging

The Packet Snooper measures the size of every packet. Over a relatively long measurement, packet size information can accumulate to a large number of samples. Reading this amount of data out of the

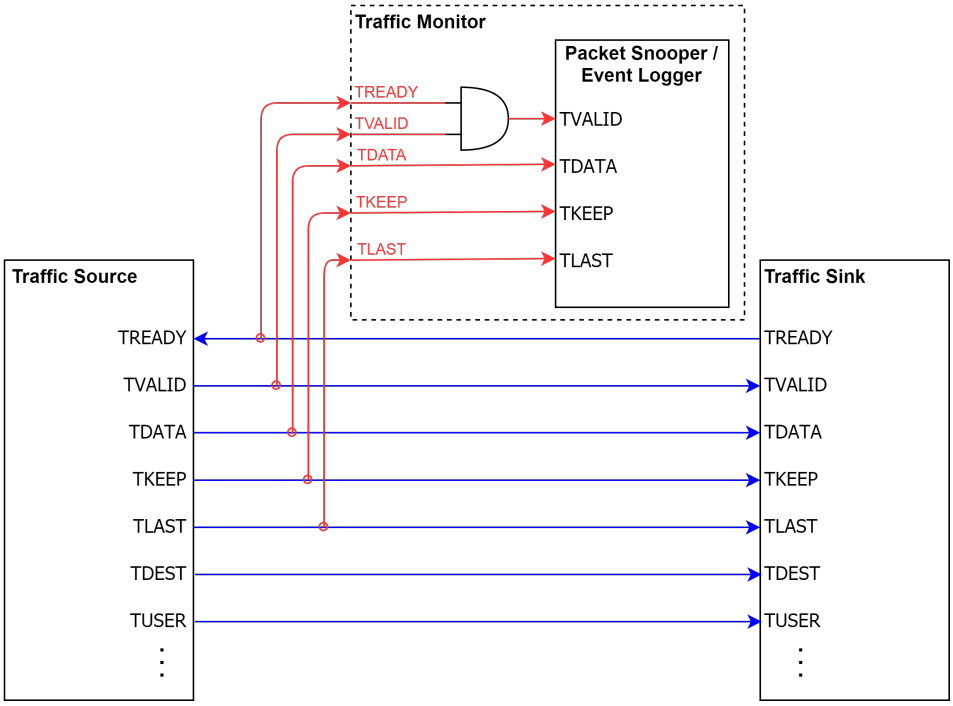

Figure 3.9: Interface of the Traffic Monitor

FPGA is normally a very resource intensive task. The EMA core can be used to turn the packet size information to a useful, much smaller amount of data. Every time the end of a packet is detected by the Packet Snooper, the size of that packet is sent to the EMA core. The EMA core will then calculate an Exponential Moving Average of the packet sizes by Equation 3.4:

$$EMA_i = (1 - \beta)EMA_{i-1} + \beta(PS_i) \tag{3.4}$$

This is similar to Equation 3.3, which was used to average latency values. In this case, EMA_i represents the current average value, EMA_{i-1} represents the previous average value, PS_i is the new packet size, and β is a smoothing factor between 0 and 1. β is an input from the user that can be changed on the fly. The closer β gets to 0, the more history is taken into account. Low values of β generally make the packet size curve smoother. On the other hand, the closer β gets to 1, the more the value of the current packet size affects the average. EMA data can then be read out at certain intervals. This interval is set by the user and can be changed on the fly.

3.5.4 Event Logging

The Event Logger provides the means to log certain events. The Event Logger of Pharos works based on the triggering technique, which is a common concept amongst many FPGA logic analyzers. Vivado's Integrated Logic Analyzer (ILA) [34] and Quartus' SignalTap [33] are examples of such systems. As discussed in Section 2.4, in the case of Vivado ILAs and Quartus SignalTap, the logic analyzer keeps sampling certain FPGA data and looks for a trigger value provided by the user. The sampled data is constantly put inside a circular buffer. The buffering stops whenever the trigger value is detected and the buffer is presented to the user.

The Event Logger of Pharos does not buffer the data. It monitors AXI-stream lines and provides a timestamp whenever the payload (TDATA) is equal to its trigger value. These timestamps become more useful when the nodes that contain the Traffic Monitor are time-synchronized using pPTP. When several FPGAs agree on a *common* time, the timestamps of a certain event can be sorted in a meaningful order, which provides a way to trace certain events (or packets) across the network. Similar to the Packet Snooper, the Event Logger has a measurement enable signal that can be controlled by the user. The duration of measurement is the time between a consecutive positive edge and a negative edge of the measurement enable. During this time, the Event Logger generates timestamps for certain events and counts the number of occurrences of that event. The timestamps and number of occurrences can be read out as separate outputs. Finally, it is worth mentioning that the trigger conditions of the Event Logger can be changed at runtime and there is no need for re-synthesis to log a new value.

Chapter 4

Performance Evaluation

This chapter discusses the performance evaluation of Pharos. We start by explaining the experimental setup and the hardware used in the experiments in Section 4.1. Then Sections 4.2 to 4.4 discuss the performance evaluation for each of the components of Pharos: pPTP, the Latency Monitor, and the Traffic Monitor. Each of these sections starts with an explanation of the methodology of the test, followed by the presentation and discussion of the results. Finally, Section 4.5 discusses the hardware usage of Pharos.

4.1 Experimental Setup

All of the experiments of this work were implemented on Xilinx Zynq Ultrascale+ XCZU19EG-FFVC1760-2-I chips [46]. This chip is part of the Fidus Sidewinder-100 [44] FPGA board. In addition to the FPGA, this chip includes a CORTEX-A53 ARM CPU [47] on the same silicon die. The Xilinx FPGA on the Sidewinder board has around 1.1 million logic units and 9.8 MB of on-chip memory. The inter-FPGA communications of the experiments were done using a 100G network and using GULF-Stream [48]. GULF-Stream is an open-source hardware IP that provides a 100G UDP link for FPGA devices. All FPGAs are connected to a Dell Z9100-ON [49] 100G network switch using 2-meter 100G copper cables.

4.2 Time Synchronization using pPTP

This section describes the evaluation of pPTP, the time synchronization protocol used in the Pharos performance monitor. The accuracy of the latency measurement and event logging of Pharos highly depend on the accuracy of time synchronization. The goal of this experiment is to determine how accurate the time synchronization protocol is. Before discussing the test setup, we must present a clear definition of accuracy. In this case, we define accuracy as the amount of time that a slave timer drifts apart from the master timer during a synchronization interval, which is precisely the amount that a slave must add to or subtract from its time to be in sync with the master time. Note that this definition does not evaluate how accurate the timers are. It is merely an assessment of whether the two pPTP instances count the same number of clock cycles during one synchronization interval. As mentioned in Section 3.2, the goal of pPTP is to make events appear as if they are happening on the same FPGA and not on separate devices. Therefore, the smaller the time difference between the master and the slave

instances is, the more accurate the measurements become. If the two FPGAs count the same number of cycles during a synchronization interval, we conclude that the measurement is cycle-accurate.

Figure 4.1 shows a synchronization process. When the slave receives the *Synchronization Response*, it calculates the network delay time using Equation 4.1:

$$T_{network} = \frac{t_{s2} - t_{s1}}{2} \tag{4.1}$$

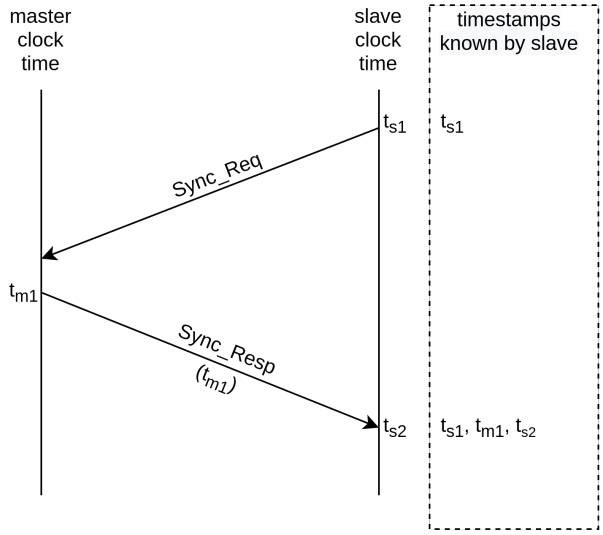

Figure 4.1: pPTP synchronization procedure

The offset between the slave and master times can then be calculated using Equation 4.2.

$$T_{offset} = |t_{s2} - t_{m1} - T_{network}| \tag{4.2}$$

Note that the slave timer can be running faster or slower than the master timer. Therefore, the offset value may become positive for some slaves and negative for others. However, since only the magnitude of the offset is of importance, Equation 4.2 represents T_{offset} as an absolute value. Note that this calculation holds the assumption of symmetrical network delays explained in Section 2.1.2.

In the remainder of this section, we describe the test setup used to characterize pPTP accuracy in Section 4.2.1, and present and discuss the results in Section 4.2.2.

4.2.1 Methodology

In this experiment, six FPGAs are deployed, one of which contains the pPTP-master and the other five contain the pPTP-slaves. Figure 4.2 pictures the test setup. In this setup, nodes 2 to 5 simultaneously synchronize their times to node 1. Each node uses a 100MHz clock and the communication between FPGAs is done at 100Gb/s using the UDP core. All of the nodes are connected to the same 100G switch using 2-meter 100G copper cables. To solely focus on the accuracy of pPTP, we ensure the network is free of any other traffic and the pPTP packets are sent in a congestion-free network.

Note that the choice of using six FPGAs is due to limitations of available hardware. The pPTP

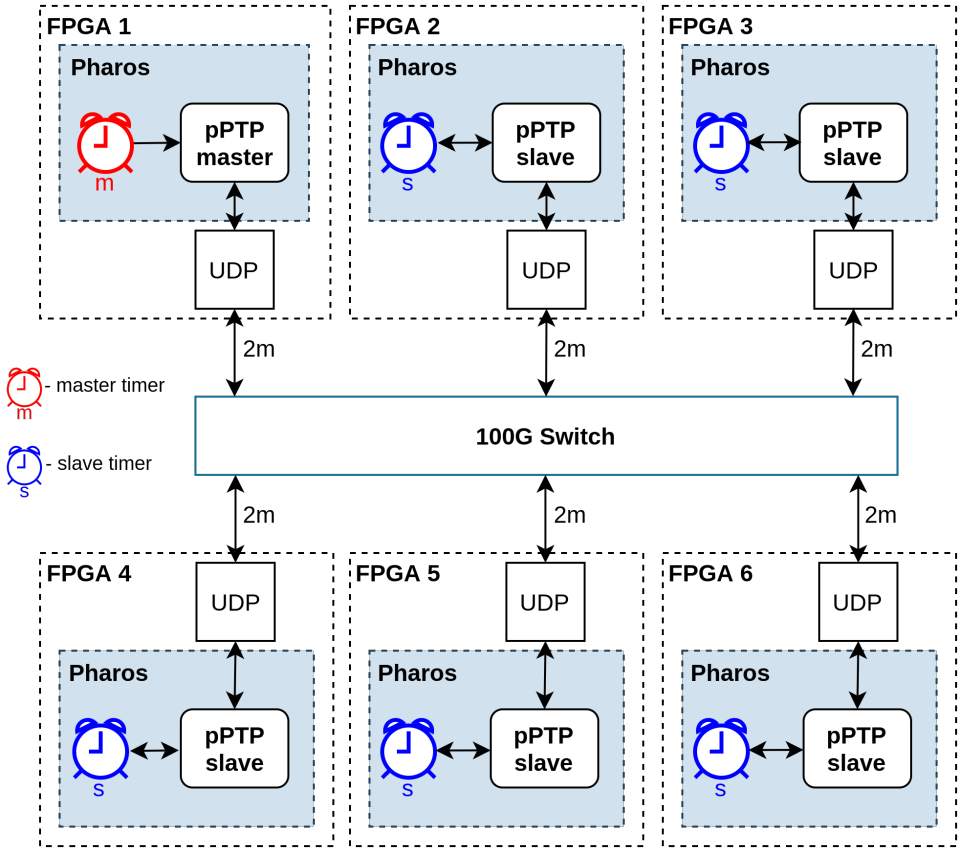

Figure 4.2: Test setup for the accuracy characterization of the time synchronization protocol deployed in Pharos.

system can be scaled to much larger networks with small cost. In the current version of Pharos, a master pPTP instance can support up to 65536 slaves using the same amount of FPGA resources (see Section 4.5 for details on hardware usage).

To analyze the effect of the synchronization interval on the accuracy of time synchronization, we sweep $Tsync$ from 100μs to 10s. Using a 100MHz clock, we go from a 10000-cycle synchronization interval to a 1-billion-cycle synchronization interval. The results are discussed in Section 4.2.2.

4.2.2 Results

Figure 4.3 shows the accuracy of time synchronization in Pharos. As mentioned in Section 4.2, the accuracy is represented by the absolute value of the time offset between the master timer and each slave timer. This essentially shows how much each slave has to shift its time at the end of a synchronization interval to be in sync with the master timer. Note that the FPGA that holds the pPTP-master instance was chosen arbitrarily. We found that some slaves have faster and some have slower clocks in comparison to the master clock. Since we are only looking at the time difference, we show the absolute value of the time offset.

Note that both axes of Figure 4.3 use a logarithmic scale. For intervals smaller than 10 ms (10^6 cycles), there was no clock drift for any of the slaves and no adjustments were needed for the slave timers. At a T_{sync} of 10 ms, the highest observed offset was 40 ns and at a T_{sync} of 1 s (1000 ms),

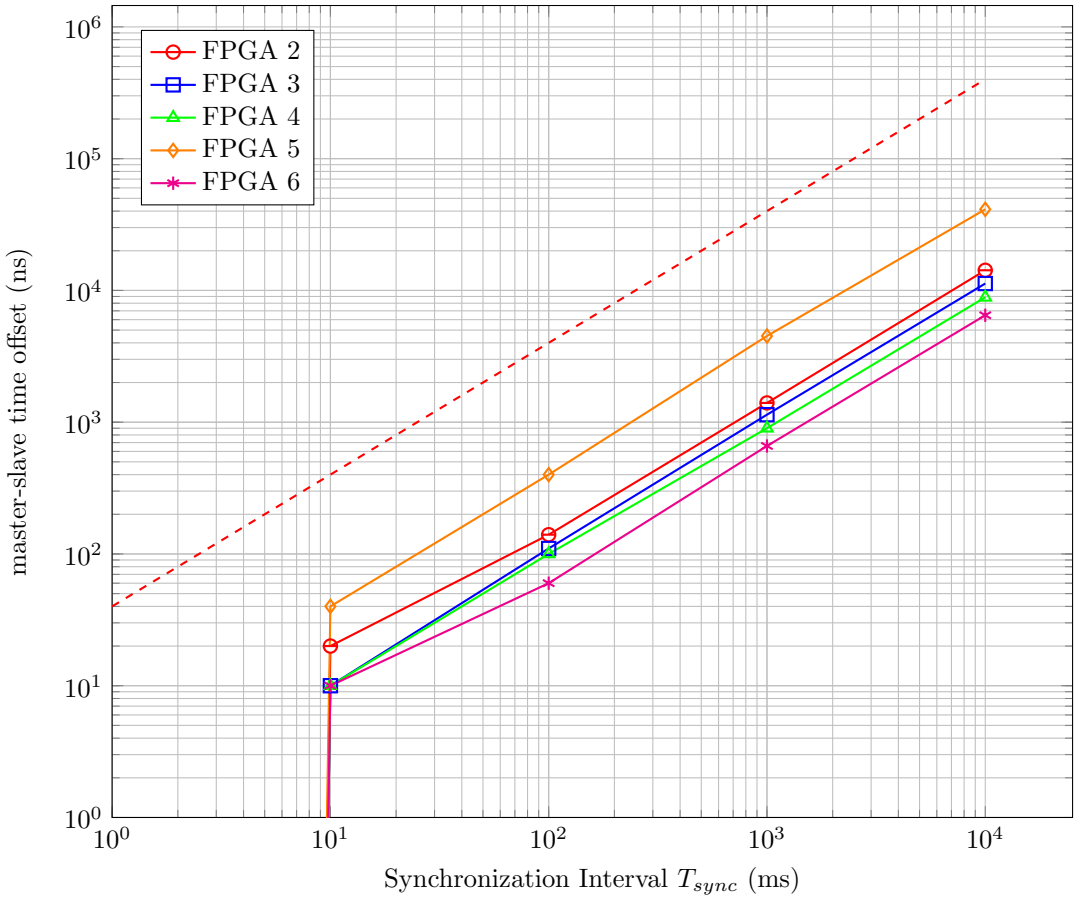

Figure 4.3: Relationship between the pPTP synchronization interval and the master-slave time offset. The dashed line indicates the maximum possible time offset due to crystal differences.

the highest observed offset was 4.51 μs ($4.51x10^3$ns or 451 cycles). Note that at 100MHz, our finest resolution of measurement is an offset of 1 cycle, equal to 10 ns.

The linear increase[1] of T_{offset} suggests that the time drift is mostly due to differences in the clock generation crystals. The reference crystal's datasheet [14] identifies a maximum 20 PPM clock drift at $25°C$. Since we are measuring an offset value between two crystals, there can be a maximum of 40 PPM clock drift. In the plot of Figure 4.3, the maximum possible clock drift due to crystal differences is shown by a dashed line. All of the FPGA lines fall within this range (below the dashed line), with the maximum drift (FPGA 5) being around 4 PPM.

We mentioned in Section 4.2 that the pPTP characterization experiments were conducted in a congestion-free network. Yet since pPTP signals are very sparse (once per T_{sync}), they have a negligible effect on the network traffic. Therefore, our results are a good indication of the accuracy of clock synchronization within Pharos. In the experiments of Pharos's Latency Monitor (see Section 4.3) we look at the overall effect of the Latency Monitor and pPTP in the presence of different amounts of network traffic.

[1]Note that this is a log/log plot, which compresses the scales, but the linearity is preserved so the lines are still linear.

4.2.3 Discussion

One potential source of error in the time synchronization protocol is the response time of the pPTP master. The response time is shown in Figure 4.4 and is denoted as t_{m_resp}. This is the time it takes for the pPTP master to respond to a Synchronization Request (Sync_Req) with a Synchronization Response (Sync_Resp). If the number of slaves connected to the master is large, the response time of the master may increase. The increase in response time only occurs if the master receives simultaneous messages from multiple slaves. During this time, the Sync_Req messages must be stored in a queue[2] (for instance, an AXI-Stream FIFO). If the master receives many Sync_Req messages simultaneously, the response time increases for some of the slaves. This introduces extra asymmetry in the network propagation calculation of Equation 4.1. This error was not seen in the experiment of Section 4.2.1 due to the small size of our experimental setup.

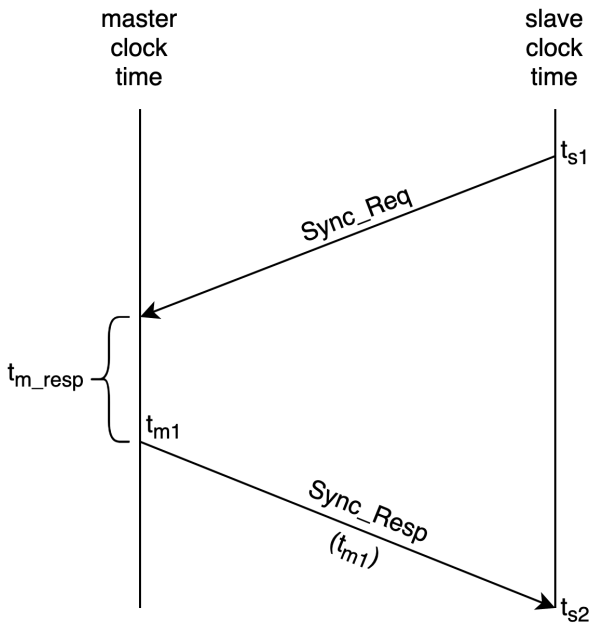

Figure 4.4: The response time of the master pPTP

Consider the following scenario: a pPTP master receives one thousand simultaneous synchronization requests from one thousand different slave instances. In the worst case, one of the requests must wait in the queue for one thousand clock cycles before the master can send a response to its associated slave instance. If the real network delay between that slave and the master is 100 cycles, then it takes 1,100 cycles for the master to see the Sync_Req message (100 cycles for the Sync_Req to reach the master's queue and 1,000 cycles for the master to read the request out of the queue). On the other hand, it only takes 100 cycles for the slave to receive the master's response. This asymmetry introduces an error of 500 cycles in the network propagation calculation of Equation 3.1, which causes the synchronization between the master and the slave to be off by an extra 500 cycles.

In the above-mentioned scenario, when requests from several slaves collide with each other, the master will respond to them one by one. This creates an offset of one cycle between the master responses to

[2]Note that the pPTP master does not have a built-in queue. This queue must be provided separately by the user. The Xilinx Vivado tools have AXI4-Stream FIFOs that can be used for this purpose.

each of the slave instances. As mentioned in Section 3.3, the next Synchronization Requests are sent one T_{sync} after the slaves receive the Synchronization Response from the master (This is also shown in Figure 3.2). Because of this, the slaves will receive the master response with an offset of one cycle and their next Synchronization Request will be one cycle apart. Therefore, it is likely that any possible collisions will be rectified after the next synchronization. It is worth noting that based on the amount of network traffic, the network propagation delay may vary from one synchronization cycle to another. This factor adds some uncertainty to the system, which may also cause random collisions to happen.

The results of Section 4.2.2 suggest that synchronization intervals smaller than 10 ms (one million cycles) result in cycle-accurate synchronizations. Therefore, we argue that if the number of slaves that connect to a master is much smaller than the synchronization interval (in terms of clock cycles), the master's response time is unlikely to introduce a significant error to the time synchronization of Pharos. For example, for a typical network size of around one thousand FPGAs, it is unlikely for a pPTP master to receive a large number of simultaneous synchronization requests (it is unlikely for all one thousand requests to reach the master pPTP at the same time if each slave sends a request every one million cycles). Moreover, if such an event happens, it is very likely that the error in the time synchronization will be rectified in the next synchronization cycle. For an error to persist in the system, the overlapping synchronization requests must reoccur in consecutive synchronization cycles over a large period of time. Nevertheless, to decrease the likelihood of this error, it is necessary for the user to ensure that the ratio of Equation 4.3 is small.

$$R = \frac{n_{slaves}}{T_{sync}} \tag{4.3}$$

In Equation 4.3, n_{slaves} represents the number of pPTP slaves connected to a single pPTP master and T_{sync} represents the average synchronization interval of the slaves in terms of clock cycles. The idea is to have a smaller number of slaves sending requests to a master over a larger amount of time. This essentially reduces the possibility of overlapping messages for each synchronization cycle. To decrease this ratio, it is best to use a hierarchical pPTP network (as described in Section 3.3.2 to limit the number of slaves that connect to each master instance. It is also possible to increase the synchronization interval. However, the latter option may affect the accuracy of time synchronization (as described in Section 4.2.2).

4.3 Latency Monitor

As mentioned in Section 3.4, Pharos generates timestamped packets for latency measurements. These packets will add to the existing traffic and may cause congestion. Moreover, the frequency at which timestamps are generated determines the resolution of the latency measurements over time. This Section explains several experiments with the goal of answering two main questions: A) How much latency does Pharos add to the system? B) What is the best latency resolution that Pharos can achieve and how much resolution does Pharos lose when network traffic increases?

To answer these questions, we conduct two experiments. In the first experiment, we compare latency measurement results of Pharos to a simple UDP loopback and look at how much latency Pharos adds to the system. In the second experiment, we sweep the network traffic and look at Pharos's best resolution in the presence of different amounts of traffic. The methodology and results of these experiments are

presented and discussed in Sections 4.3.1 and 4.3.2.

4.3.1 Methodology

Figure 4.5 shows the test setup for this experiment. To determine the insertion latency of Pharos, we compare a simple round-trip time of packets to a unidirectional latency measurement using Pharos's packet generation and time synchronization. In both cases, We use GULF stream [48] as the 100G UDP core and the two FPGAs are connected to a 100G switch using 2-meter cables. Apart from the UDP core, the rest of the FPGA applications run at 100MHz.

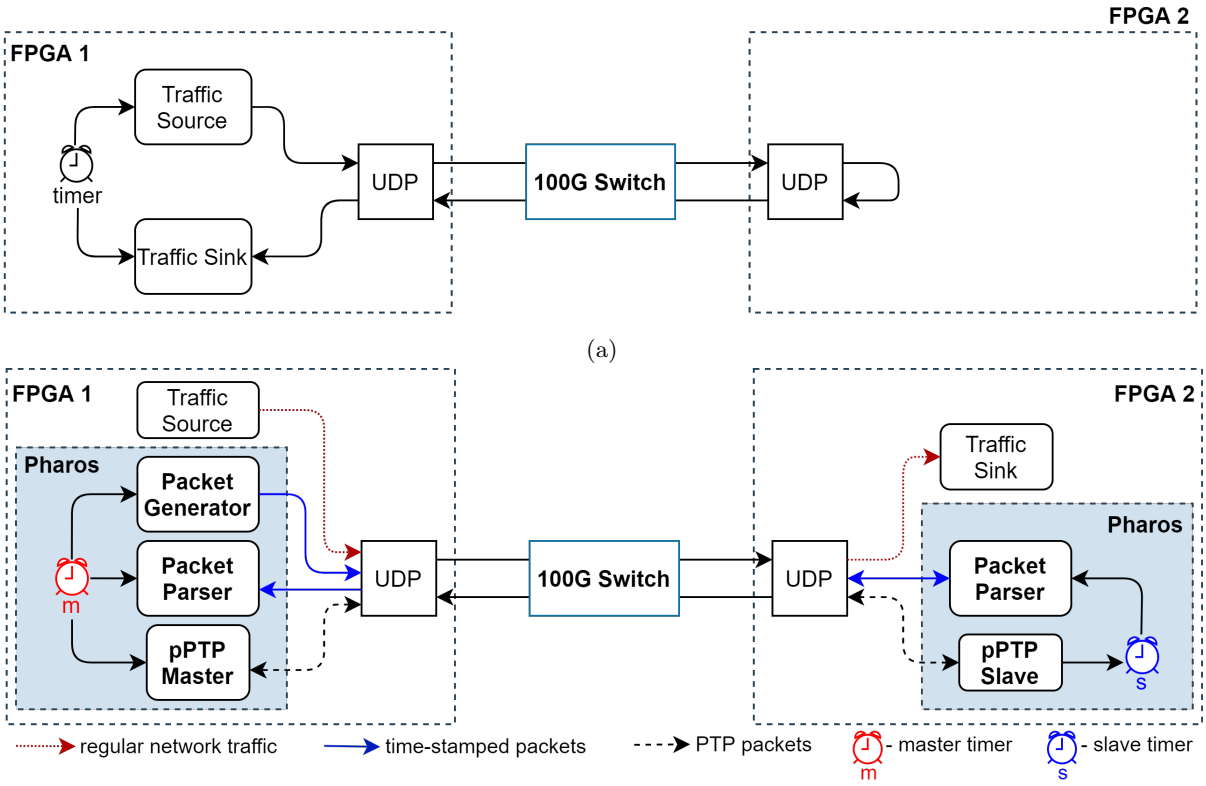

Figure 4.5: Test setup for latency measurements. (a) latency of UDP loopback (b) Latency measurement using Pharos

Figure 4.5a shows a simple UDP loopback, where packets are streamed from FPGA 1 through the 100G network to FPGA 2 and then immediately sent back to FPGA 1. All of these packets are timestamped at the Traffic Source. The latency of each packet is measured when they arrive at the Traffic Sink using the same timer. On the other hand, Figure 4.5b uses Pharos's Latency Monitor. The pPTP master and the master timer are placed in FPGA 1. FPGA 2 holds the pPTP slave and the slave timer. We fixed the synchronization interval at 1 ms. As shown in Section 4.2.2, this is the largest interval that ensures maximum possible accuracy for pPTP. Throughout the test, master-slave offset (Equation 4.2) is monitored to make sure unidirectional latency measurements are always cycle-accurate at this T_{sync}.

In the case of Figure 4.5b, timestamped packets are generated at the Packet Generator within

FPGA 1. Latency is then measured at two stages. The first measurement is done at the Packet Parser of FPGA 2. This represents the unidirectional latency measurement. Then this Packet Parser forwards the packets back to the Packet Parser of FPGA 1 where a round trip latency measurement is obtained. This has two purposes:

1. The round-trip time presents a better comparison to the case of UDP loopback of Figure 4.5a and removes pPTP's inaccuracy in the assumption of symmetric trip times
2. A comparison between unidirectional and round-trip latency measurements is used to verify the accuracy of latency measurements

In the experiment of Figure 4.5b, the Traffic Source is used as a fake application to generate traffic using all of the bandwidth. This is done to test the insertion latency of Pharos in very high traffic. To find the highest possible traffic, all Pharos packet generations are stopped (both the Packet Generator and pPTP) and the Traffic Source generates packets of size 64B. At 100 MHz, this becomes 51.2 Gb/s. The UDP core then adds 8 bytes of UDP header, 20 bytes of IPV4 header, and 14 bytes of Ethernet header to each packet. Therefore, the AXI payload with the addition of the headers does not fit in one Ethernet packet and each payload has to be sent with 2 packets (128B). Hence, at 100 MHz packet generation, we are sending 102.4 Gb/s, which exceeds the 100G limit. At this rate, the network hardware begins applying back-pressure and finally packet drops and huge latency spikes are observed. To avoid packet loss, we slowly decreased the packet generation rate of the Traffic Source until no packet loss occurred. This happened at 98.304 Gb/s, which we will refer to as the 100G line rate. At this traffic rate, we enabled Packet Generator and pPTP for the measurement of insertion latency (See Appendix A for more details).

To measure the resolution of the Latency Monitor, we used the test setup of Figure 4.5b with one difference. In this case, we swept the bandwidth usage of the Traffic Source from 0 Gb/s to a maximum of 98.304 Gb/s and recorded the highest possible frequency of latency measurements (frequency of timestamped-packet generation within Pharos) that did not result in packet loss and latency spikes.

4.3.2 Results

The round-trip latency explained in the base setup of Figure 4.5a was averaged over a 1 minute period. This measurement was conducted at 100G line rate traffic (non-Pharos traffic using 98.304Gb/s). As we explained in Section 4.3.1, this was the highest bandwidth usage that did not result in network back-pressure and packet loss. The average base latency over a 1 minute period was measured to be $2.140\mu s$.

Table 4.1: Insertion latency of Pharos at full bandwidth

T_{gen} (μs)	6.70	10	100	1000	10000
L_{rt} (μs)	2.162	2.147	2.146	2.146	2.146
L_{uni} (μs)	1.077	1.069	1.070	1.070	1.070
L_{ins} (ns)	22	7	6	6	6

Table 4.1 shows the results for the experiment of Figure 4.5b. In this table, T_{gen} represents Pharos's packet generation period, while L_{rt} and L_{uni} represent round-trip latency and unidirectional latency, respectively. L_{ins} is the insertion latency of Pharos (L_{rt} - 2.140) and is shown in units of nano-seconds.

All of these results are average values over a 1-minute period and were obtained when the Traffic Source was using 98.304% of the bandwidth. In these conditions, the lowest possible sampling period for latency measurements was found to be 6.70μs. This means that at very high traffic, Pharos can send timestamped packets every 6.70μs without causing network congestion. Any lower T_{gen} will cause packet loss and huge latency spikes. As we increased T_{gen} (decreased sampling frequency), the latency converged to 2.146 μs and the insertion latency of Pharos converged to 6 ns. Table 4.1 is also an indication of the accuracy of latency measurements by showing a comparison between the round-trip and unidirectional latency values. Pharos's unidirectional latency values are on average around 0.3% higher than half the round-trip latency values. This difference is because of pPTP's assumption of symmetrical delays (see Section 2.1.2). Figure 4.6 also shows the insertion latency of Pharos when the network is at 100G line rate. The insertion latency spikes when the packet generation period is lower than 6.70μs. For higher T_{gen} values, the insertion latency converges to 6ns. As previously mentioned, this value is an average value over a one-minute period. This means that the insertion latency of Pharos at an operating frequency of 100 MHz is on average below 1 clock cycle.

Figure 4.7 is a histogram of all latency samples for both base setup (Figure 4.5a) and Pharos (Figure 4.5b). Both histograms contain the latency values that were collected over a 1 minute period. For the base case, every packet was timestamped at 100G line rate; therefore, 5.76 billion latency numbers were collected over 1 minute. For the case of Pharos, the Traffic Source occupied 98.304% of the bandwidth and latency was collected at intervals of 6.70μs, resulting in about 8.94 million samples. Because of this difference in the number of data points, the histograms show what *percentage* of data falls within each latency bin. These histograms confirm that no latency spikes were observed when sampling latency using Pharos at intervals of 6.70μs at a very highly congested network. In the base case, all latency samples fell into a 30ns range, while the Pharos latency measurements had a slightly wider distribution within a 100ns range. This can be explained by the fact that the Pharos latency monitor and pPTP cores occasionally add traffic to a network that is already operating at line rate. This causes packets to be temporarily stored in the FIFOs within the UDP core, adding a bit of extra latency to the system.

Finally, Figure 4.8 shows the best resolution achievable by Pharos in the presence of network traffic. In this plot, resolution is defined as the lowest possible interval between two consecutive latency measurements. Lower intervals are more desirable since collecting more latency samples results in a better resolution. In this test, we swept the bandwidth used by the Traffic Source from 0Gb/s to 98.304Gb/s (100G line rate), and found the lowest possible sampling interval (highest possible sampling rate) that did not result in packet loss. At line rate traffic, the lowest achievable sampling period is 6.7μs. At 95% of the line rate, the sampling interval drops to less than 250ns and at 90%, this number drops to less than 100ns. When network traffic is using half of the 100G bandwidth, Pharos can sample latency every 20ns (every two cycles at 100MHz), which is the lowest possible measurement interval for Pharos. Note that Pharos does not have to operate at these rates and the rates simply represent the highest possible sampling frequency in the presence of different amounts of traffic. (See Appendix A for more details)

4.4 Traffic Monitor

As explained in Section 3.5, the Traffic Monitor is not a pass-through system, meaning that none of the traffic that is being monitored passes through Pharos. Therefore, by design, the Traffic Monitor cannot back-pressure on the passing traffic and does not add any latency to the system. In this section we focus

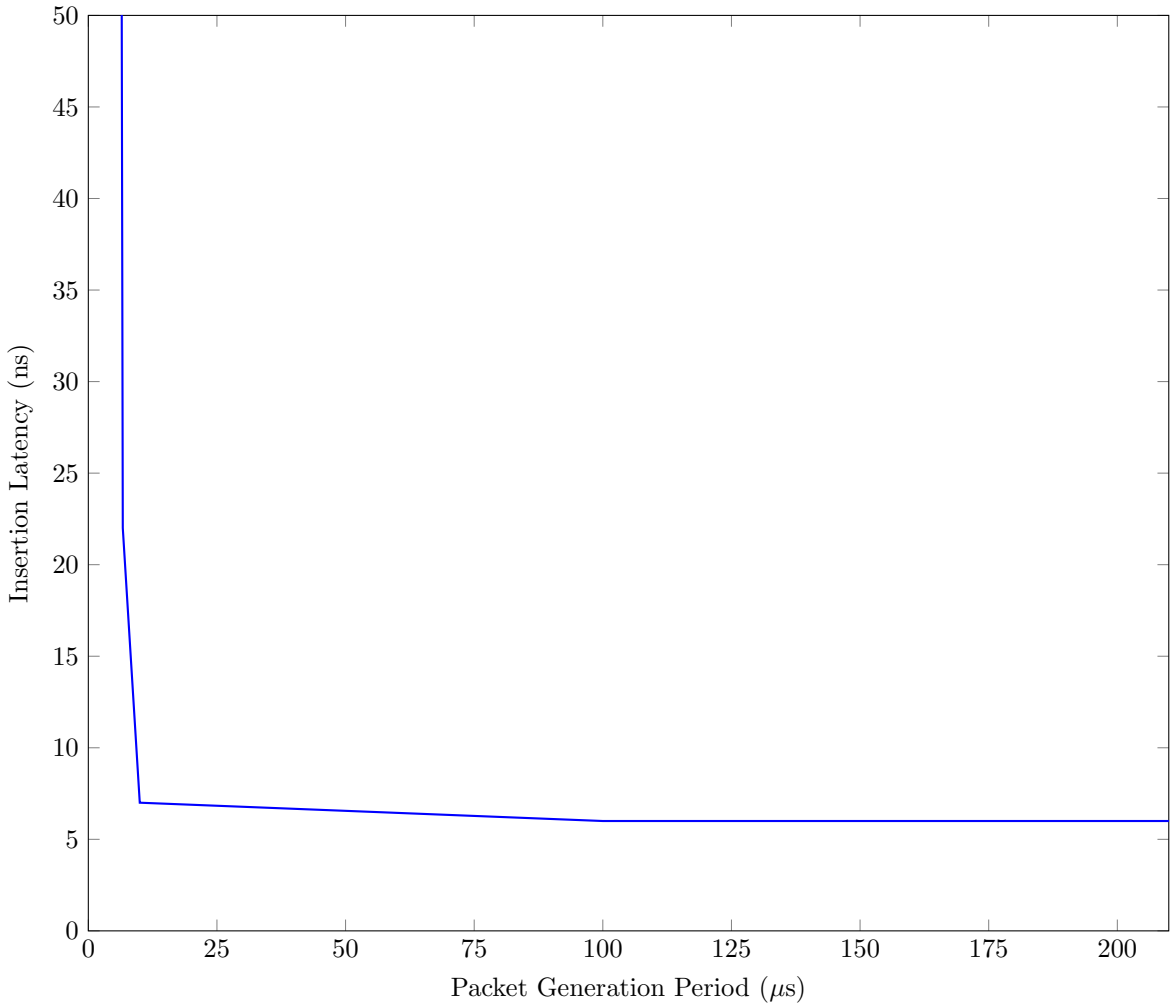

Figure 4.6: Relationship between packet generation period and insertion latency of Pharos at 100G line rate

on the functionality and accuracy of the Traffic Monitor.

4.4.1 Methodology

Figure 4.9 shows the experiment setup. In this experiment, we used three FPGAs, and synchronized their times using pPTP. We placed the master timer and the pPTP-master in FPGA 2 (an arbitrary choice, any other FPGA could have been chosen to hold the master instance). FPGAs 1 and 3 synchronize their times to FPGA 1 at intervals of 1 ms. At this rate, pPTP guarantees all timers to be cycle-accurate (see Section 4.2.2). The goal of this experiment is to send 1 million packets from FPGA 1 to FPGA 2 and forward only a fraction of these packets to FPGA 3 and monitor all traffic in every step.

FPGA 1 - The Traffic Generator

The Traffic Generator generated 1 million packets. The payload (TDATA channel) of these packets incrementally increased from 1 to 1 million (the first packet's payload was 1, the second packet was 2,

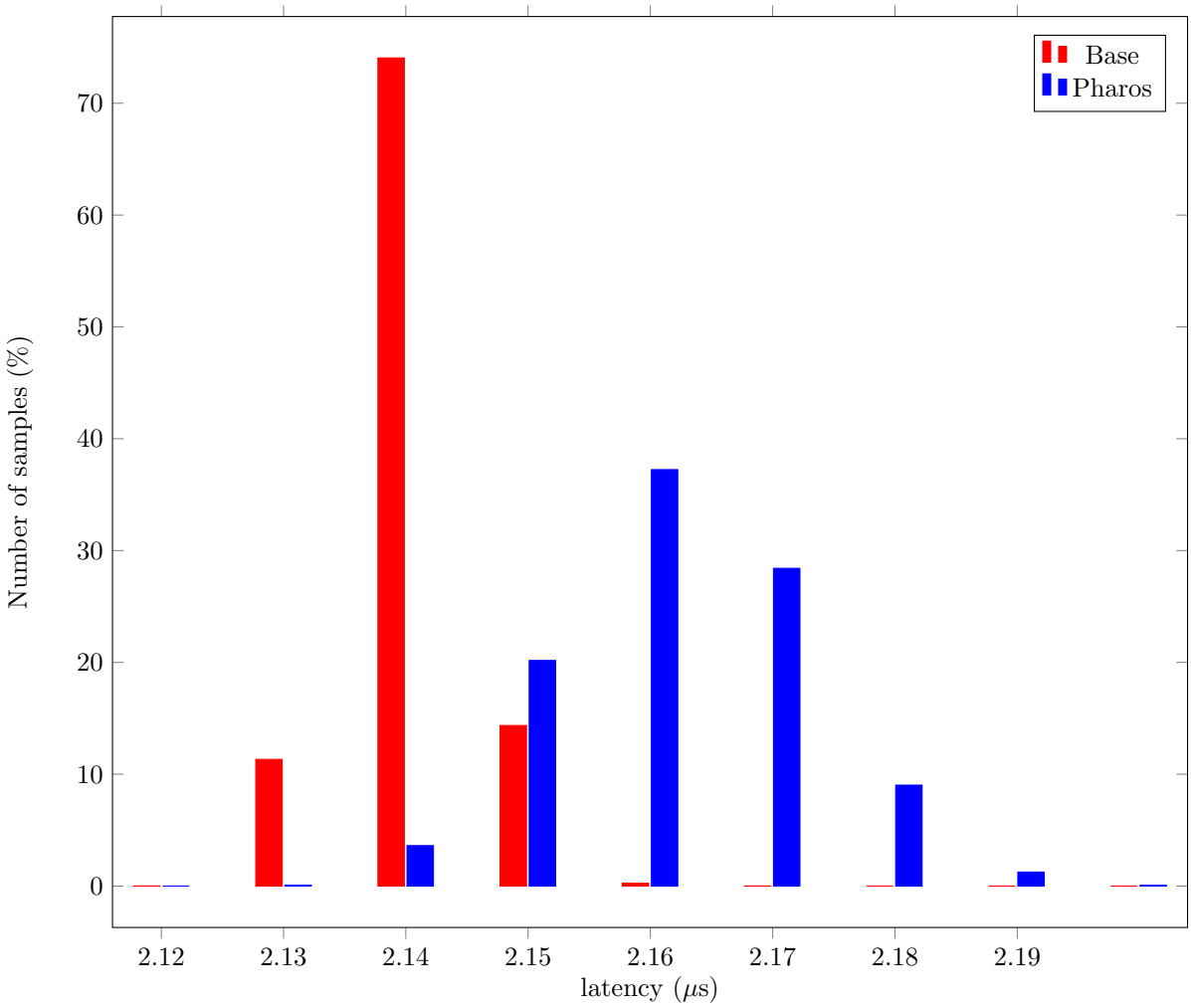

Figure 4.7: Histogram of latency values

and so on. The one-millionth packet's payload was 1,000,000). The packets were of four different sizes. The first quarter of packets (first 250,000 packets) were of size 64B. The next three quarters were 128B, 192B, and 256B, respectively. The packets were sent to FPGA 2 over the 100G network. To avoid congesting the network, one flit was generated every 4 cycles. This means that smaller packets took less time to be sent. For example, packets of size 64B would fit into one flit and could be sent every 4 cycles. However, packets of size 128B would have to be sent using 2 flits and would take 8 cycles to be sent. Similarly, packets of size 192B and 256B would take 12 and 16 cycles. One Traffic Monitor is used to snoop on the outputs of the Traffic Generator.

FPGA 2 - The Traffic Forwarder

Upon receiving the packets, the Traffic Forwarder forwarded only the ones that had payloads greater than 320,000 to FPGA 3. This number was chosen arbitrarily. Therefore, none of the packets with size 64B get forwarded to FPGA 3. Two Traffic Monitors are used to monitor the inputs and outputs of the Traffic Forwarder.

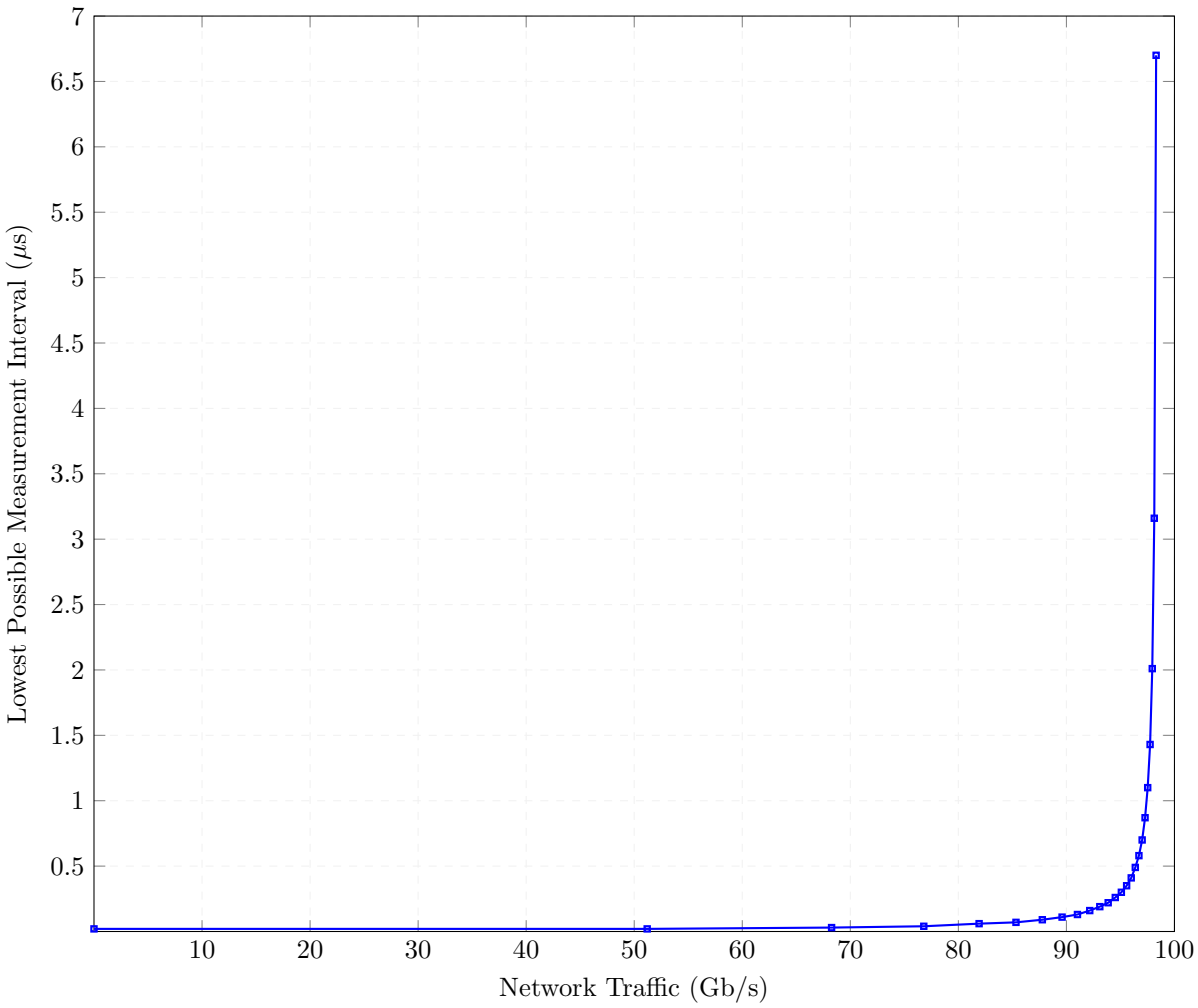

Figure 4.8: Relationship between the lowest possible sampling period of latency in Pharos and 100G network traffic (traffic used by the rest of the network)

FPGA 3 - The Traffic Sink

Traffic Sink only receives the packets from FPGA 2. One Traffic Monitor is used to snoop on its inputs.

Event Logging

Each Traffic Monitor contained a Packet Snooper, an EMA core, and two Event Loggers. The trigger values of the two Event Loggers within each FPGA were set to 452,000 and 70. These two values were chosen arbitrarily but in a way that one of them is smaller and one is larger than 320,000. This means that the packet with the payload of 452,000 travels from FPGA 1 to FPGA 3, but the packet with the payload of 70 only travels from FPGA 1 to FPGA 2 and stops at the Traffic Forwarder.

Control Signals and Data Collection

To synchronize the start and end of measurement for all three FPGAs, an enable signal was sent from FPGA 1 to FPGA 2 a cycle before sending the first packet. Similarly, a stop signal was sent 1 cycle after sending the 1 millionth packet from FPGA 1 to FPGA 2. The second FPGA then forwarded both of these signals to FPGA 3 immediately upon receiving them. The start/stop signals were used to start and end the measurements for all monitors. The results of each FPGA were read out by the ARM core of the same Sidewinder board, where a host software on each processor summarized the results for each monitor into a log file. These results are discussed in Section 4.4.2.

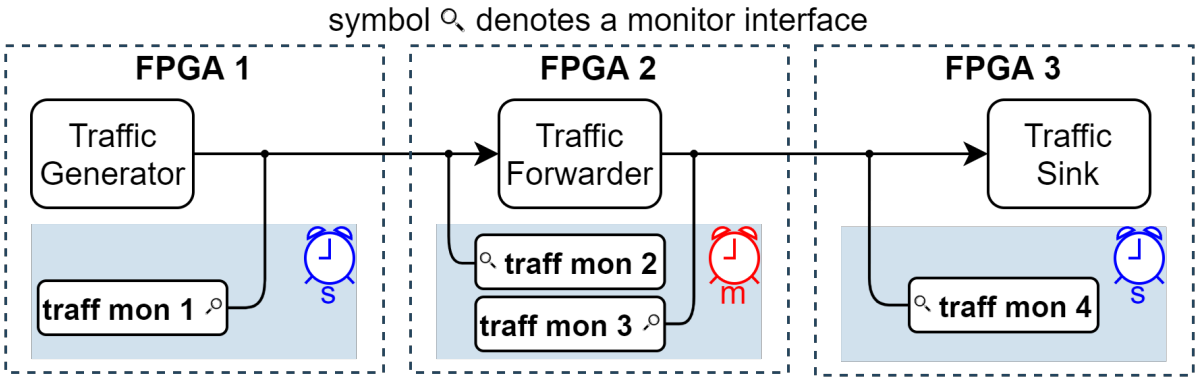

Figure 4.9: Test setup for traffic monitoring system. The red timer denotes the master timer and the blue clocks are slave timers.

4.4.2 Results

Listings 4.1 to 4.4 show the log files from all four Traffic Monitors, shown in Figure 4.9. Monitors 1 and 2 both observed 1 million packets during about 10 million cycles (Packets of size 1 flit were sent over one million cycles, packets of size 2 flits over two million cycles, packets of size 3 flits over three million cycles, and packets of size 4 flits over four million cycles). However, monitors 3 and 4 both observed 680 thousand packets during the same period. This was because the Traffic Forwarder dropped the first 320 thousand packets. For the same reason, the Event Loggers of monitors 1 and 2 output a timestamp for packets with their payload equal to 70, while monitors 3 and 4 did not observe such packets. However, all four monitors generated a timestamp for their second trigger value (452,000). These timestamps are in chronological order from monitor 1 to monitor 4. The timestamps are in terms of cycles of 10 ns. This means that a packet with a payload of 452,000 was seen in the second Monitor 102 cycles after it was observed in Monitor 1. Then Monitor 3 observed this packet only 2 cycles after Monitor 2 (it only went through the Forwarder during this time). Finally, Monitor 4 observed this packet 103 cycles after Monitor 3.

The Monitors also collected packet sizes during the measurement. The output frequencies of the EMA cores were set to 1 million cycles. Therefore, during a measurement period of 10 million cycles, 10 samples were collected. Each EMA output in the logs represents one measurement. As mentioned in Section 4.4.1, one flit (64B) was sent every four cycles. Therefore, larger packets were sent over a longer period of time and more EMA samples were collected for larger packets. The logs indicate an increase in

the size of packets from 64 bytes to 256 bytes. Since the first 320,000 packets were not sent from FPGA 2 to FPGA 3, Monitors 3 and 4 show 0 for their first measurement.

Listing 4.1: traff_mon 1

```
cycles:  10000002
packets: 1000000

EMA (B):
64,128,128,192,192,192,256,256,
256,256,

trigger on 70
timestamps: (1)
211264944231,

*****

trigger on 452000
timestamps: (1)
211267559954,
```

Listing 4.2: traff_mon 2

```
cycles:  10000002
packets: 1000000

EMA (B):
64,128,128,192,192,192,256,256,
256,256,

trigger on 70
timestamps: (1)
211264944333,

*****

trigger on 452000
timestamps: (1)
211267560056,
```

Listing 4.3: traff_mon 3

```
cycles:  10000002
packets: 680000

EMA (B):
0,128,128,192,192,192,256,256,
256,256,

trigger on 70
timestamps: (0)

*****

trigger on 452000
timestamps: (1)
211267560058,
```

Listing 4.4: traff_mon 4

```
cycles:  10000002
packets: 680000

EMA (B):
0,128,128,192,192,192,256,256,
256,256,

trigger on 70
timestamps: (0)

*****

trigger on 452000
timestamps: (1)
211267560161,
```

4.5 Hardware Usage

Table 4.2 summarizes the resource utilization of Pharos and its components as reported by Vivado synthesis. The master and slave components of pPTP are combined into one entry since they use a small number of resources, even though they are to be placed on different FPGAs. The Traffic Monitor entry includes one Packet Snooper, one Event Logger, and one EMA core. The Latency Monitor entries show the resources used by the combination of one Packet Generator and one Packet Parser. All of the components of Pharos use a constant amount of resources except the Packet Parser. The resource utilization of the Packet Parser changes depending on the number of nodes it can support (based on setting a pre-synthesis parameter). As the size of the Packet Parser fan-in increases, the number of LUTs, FFs, and BRAMs that it uses increases linearly (see Figures 4.10 and 4.11). Table 4.2 shows the number of resources used by the Packet Parser with fan-ins of size 1000, 10000, and 30000. These numbers are shown only as examples of three different sizes. In all of these three rows, the Packet Generator only uses a very small portion of the resources (122 LUTs, 65 FFs, and none of the block RAMs).

Table 4.2: Resource Utilization of Pharos and selected components as reported by Vivado systhesis

Component	**LUTs**	**FFs**	**BRAMs**
pPTP + timers	996	325	0
Traffic Monitor	529	265	0
Latency Monitor - 1000	1,122	6,197	5
Latency Monitor - 10000	53,445	41,092	50
Latency Monitor - 30000	158,437	121,095	141
Available in Sidewinder	522,720	1,045,440	986

Figure 4.10 shows the relationship between the fan-in size of the Packet Parser and the number of FFs and LUTs used by the Packet Parser. Similarly, Figure 4.11 presents the number of block RAMs used by Packet Parsers of different fan-in sizes. The fan-in sizes in these Figures span from 32 (TDEST width of 5) to 65,536 (TDEST width of 16), which is the largest possible fan-in size in the current version of Pharos. These two Figures clearly show the linear relation between the size of the Packet Parser and its resource utilization. The horizontal dashed lines in these Figures show the available resources in the Fidus Sidewinder board. These lines are put in to provide a point of reference with regards to how much of the Sidewinder resources are used for each Packet Parser size. Note that the y-axis of Figure 4.10 is in units of 1 million logic units and the x-axes of Figures 4.10 and 4.11 are both in units of 10 thousand. Appendix C presents the detailed number of resources used by Packet Parsers with different fan-in sizes.

Table 4.3: Resource Utilization of Pharos shown as percentages on several different FPGA boards

FPGA Board	**FPGA Family**	**LUTs %**	**FFs %**	**BRAMs %**
VU19 [50]	Virtex 6	0.064	0.083	0.237
Alpha Data 8k5 [51]	Kintex Ultrascale	0.399	0.512	0.237
Fidus Sidewinder [44]	Zynq UltraScale+	0.506	0.649	0.507
Zedboard [52]	Zynq 7000	18.382	23.566	10

Table 4.3 shows the resource utilization of Pharos in terms of percentages on several different FPGA boards. The FPGAs are chosen to represent a spectrum of high-end, mid-range, and low-end FPGAs from different Xilinx FPGA families. The VU19 [50] is a high-end FPGA from the Virtex 6 Family of FPGAs. The Zedboard [52] is a low-end FPGA from the Zynq 7000 FPGA family. The Fidus Sidewinder board that was used for the implementation of Pharos in the experiments of this thesis has a mid-range ZU19EG [45] FPGA from the UltraScale+ family. As previously mentioned, the resource utilization of the Packet Parser changes depending on the size of its fan-in. For Table 4.3, we fixed the fan-in size of the Packet Parser to be 1000. This number represents a reasonable FPGA cluster size[3]. This means that the FPGA can conduct latency measurements between itself and up to 1000 other FPGAs. In this case, the total number of resources used by Pharos comes to 2,647 LUTs, 6,787 FFs, and 5 BRAMS.

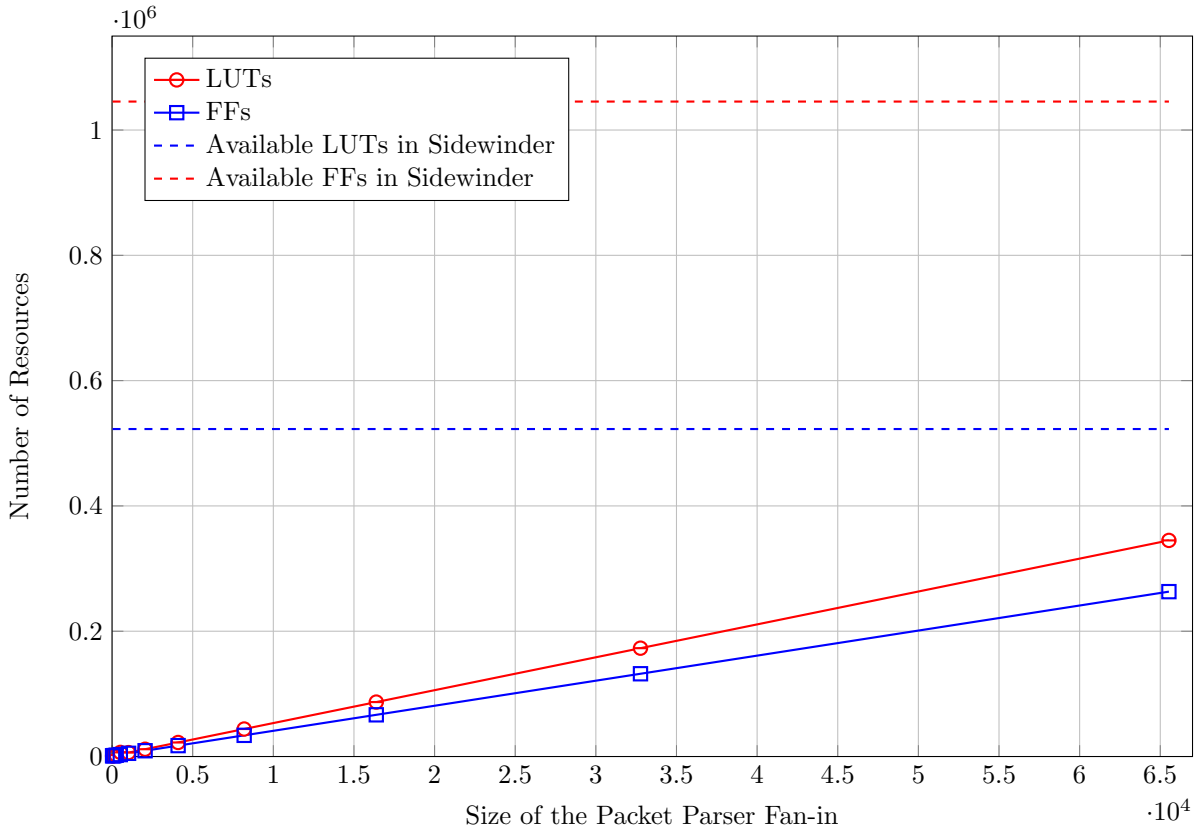

Figure 4.10: Relationship between the size of Packet Parser fan-in and the resource utilization of the FPGA in terms of look-up tables (LUTs) and flip-flops (FFs). The x-axis is in units of 10k and the y-axis is in the units of 1M.

[3]The Microsoft Catapult project[1] deployed a cluster of 1,632 machines.

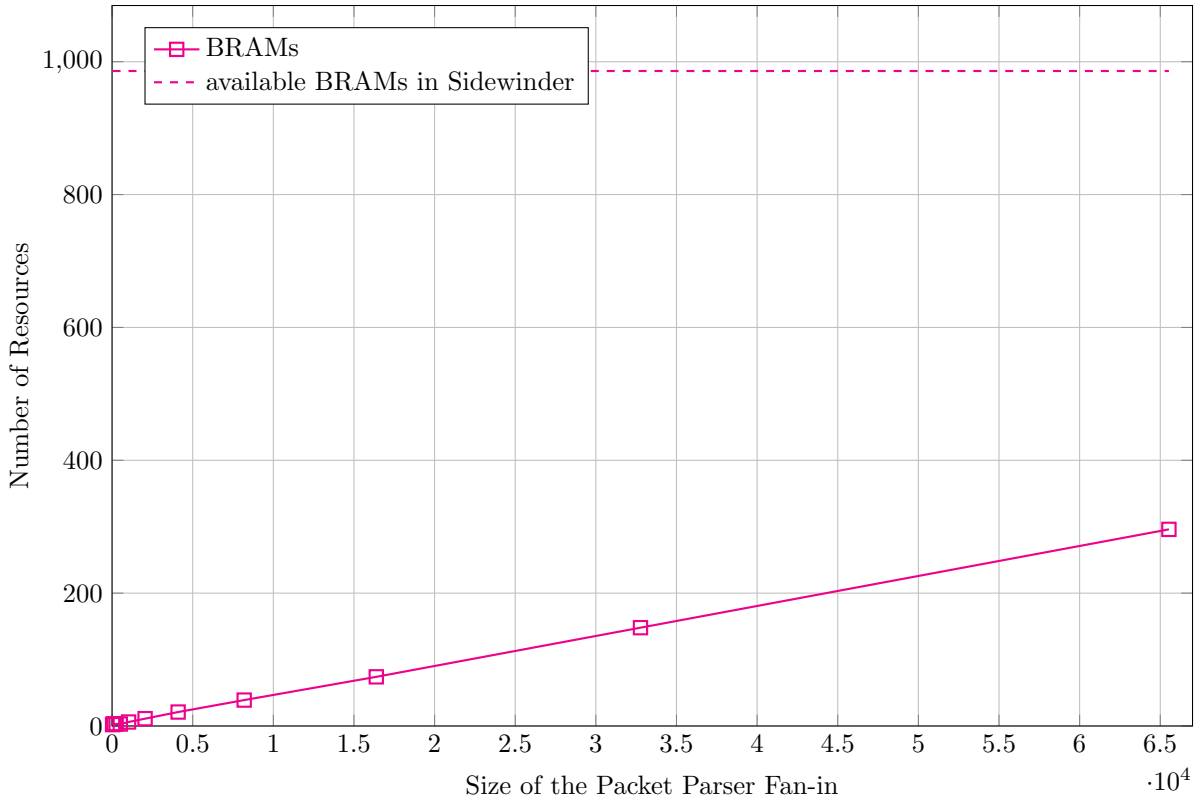

Figure 4.11: Relationship between the size of Packet Parser fan-in and the resource utilization of the FPGA in terms of block RAMs (BRAMs). The x-axis is in the units of 10k.

Chapter 5

Conclusion

In this thesis, we presented Pharos, a performance monitor for multi-FPGA systems. In this final chapter we examine a summary of the contributions of this thesis, present a final conclusion, and discuss the future work.

5.1 Summary of Contributions

This thesis presented Pharos, a performance monitor for multi-FPGA systems. The main contributions of Pharos are:

- pPTP: A time synchronization protocol based on the Precision Time Protocol (PTP) for multi-FPGA systems
- Latency Monitor: A latency monitoring system capable of cycle-accurate unidirectional point-to-point latency measurements within a multi-FPGA system
- Traffic Monitor: A traffic monitoring system capable of monitoring internal traffic of the FPGA and throughput measurements of AXI-Stream lines
- Event Logger: An event logging system capable of triggering on user-specified conditions and providing cycle-accurate timestamps of those events

Pharos uses the idea of *global time* to synchronize the time of various nodes to a common time. The time synchronization protocol used in Pharos is a modified version of the Precision Time Protocol (PTP), customized to fit a multi-FPGA environment. This time synchronization protocol, which we have named pPTP, defines a master-slave relationship between all nodes of the datacenter to establish time synchronization. Pharos is compatible with hierarchical structures, where time synchronization is done in a tree-like structure. In this scheme, one node is chosen as the *GrandMaster clock*, which provides the reference clock. The rest of the network nodes synchronize their timers to the GrandMaster clock in a hierarchical structure, where nodes in each level establish a master-slave relationship with the nodes in the next level. In pPTP, each slave can choose to synchronize its time with the master clock at a frequency specific to that slave and based on its timing requirements. Our experiments demonstrate that pPTP is capable of cycle-accurate time synchronization with synchronization intervals as large as 10 milliseconds.

Pharos is capable of cycle-accurate unidirectional point-to-point latency measurements. The timestamping of the Latency Monitor uses an AXI interface and is independent of the lower-level network protocols. We tested Pharos in a 100G network and showed that at line rate we can collect latency measurements at a rate of $6.7\mu s$ per sample. The sampling interval drops to less than 250ns and less than 100ns at 95% and 90% of the line rate, respectively. The Latency Monitor with the addition of pPTP synchronization, at its worst case (when the network traffic is at line rate of 98.304 Gb/s and packet generation is at $6.7\mu s$), adds less than 30ns of overhead latency to the network.

The traffic monitoring system of Pharos is capable of monitoring internal AXI lines of FPGAs without any overhead latency. The Traffic Monitor is able to snoop on AXI4-Stream lines and collect network information such as the number of passing packets, throughput, and average packet size. The Traffic Monitor includes an event logging system, which is able to trigger on user-specified conditions and generate a timestamp whenever the conditions are met. Combined with the network time synchronization using pPTP, Pharos provides the ability to trace events within the network.

5.1.1 Discussion

The main idea that initiated this project was a way to determine the sequence of events in a multi-FPGA setup. This was only possible if all FPGAs agreed on a common time. The choice of the PTP protocol for multi-FPGA time synchronization proved to be a successful one, as it enabled cycle-accurate synchronization between many FPGAs. Implementing the modified version of this protocol for FPGAs removed the need for extra hardware and added some programmability to the protocol, such as customizing the synchronization frequency for each slave. The time synchronization laid the foundation to implement more useful functionalities such as unidirectional latency measurement and event logging.

Overall, our attempt to implement instrumentation to monitor FPGA performance was successful. All of the tools of Pharos are light in terms of their resource utilization. In addition, Pharos is not costly in terms of its overhead latency. Moreover, being independent of the underlying network protocols makes Pharos easily transportable from one network to another. Lastly, Pharos is made to be compatible with Galapagos [53], which makes its future integration into Galapagos easier. Integration of Pharos into Galapagos will enable monitoring heterogeneous systems.

5.2 Future Work

This section explores some of the possibilities and potential directions for future work on Pharos. This includes making Pharos compatible with heterogeneous networks, developing a software layer with potential visualization tools, and expanding the capabilities to a wider range of functionalities.

5.2.1 Compatibility with Heterogeneous Systems

One of the most interesting areas of improvement is making Pharos compatible with heterogeneous systems that deploy CPUs and GPUs, as well as FPGAs. Various research has been done on using FPGAs in heterogeneous systems. Galapagos [53] is one of the best examples. Galapagos provides users the ability to use heterogeneous systems without worrying about the underlying network and communication protocols. Pharos is designed to be compatible with Galapagos and can be integrated into Galapagos to be used in a heterogeneous environment.

5.2.2 Improvements and Enhancements to the Time Synchronization

Another possible area of improvement is the time synchronization protocol. One particular point of enhancement is dynamic adjustment of slave synchronization intervals (T_{sync}). This means that given a threshold of tolerable error, each slave can dynamically adjust its T_{sync}. This helps pPTP slaves adapt to different network conditions such as different clock generation crystals, network speeds, cable lengths, and the amounts of network traffic at any given time.

5.2.3 Expanding to Other Communication Protocols

The Traffic Monitor's capability can be extended to monitor other AXI protocols. The capability of monitoring AXI-Lite and memory mapped AXI protocols such as AXI-Full enables Pharos to be used in a wider range of applications, including monitoring on-chip memory communications.

5.2.4 Software Layer and Visualization

Lastly, development of a unified software would help users analyze network data more easily. One of the most important capabilities of the software layer can be reading and combining Traffic Monitor logs from different nodes into a unified log. Another potential capability of the software can be visualization of the network traffic. This can be implemented by reading the output of the latency monitor continuously and providing a live visualization of how much traffic is within each region of the cluster.

Bibliography

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". *IEEE Micro*, 35(3):10–22, 2015.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. "A cloud-scale acceleration architecture". *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[3] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. "Enabling FPGAs in Hyperscale Data Centers". *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086, 2015.

[4] Amazon EC2 F1 Instances. pages 1078–1086, 2015. [Online] Available: aws.amazon.com/ec2/instance-types/f1/.

[5] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack". *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, 2014.

[6] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center". *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 237–246, 2017.

[7] S. A. Fahmy, K. Vipin, and S. Shreejith. "Virtualized FPGA Accelerators for Efficient Cloud Computing". *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435, 2015.

[8] Arzhang Rafii. Pharos: a Multi-FPGA Performance Monitor. 2020. [Online] Available: https://github.com/UofT-HPRC/pharos.

[9] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)*, 2019.

[10] Xilinx. "AXI Reference Guide". 2017. [Online] Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[11] D. L. Mills. "Internet time synchronization: the network time protocol". *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[12] Richard Bellman. "On a routing problem". *Quarterly of Applied Mathematics*, pages 87–90, 1958.

[13] L. R. Ford. "Network flow theory". 1956.

[14] Renesas Electronics. "25MHz 3.2 x 2.5mm Reference Crystal DataSheet". 2019. [Online] Available: https://www.idt.com/us/en/document/dst/603-25-150-datasheet.

[15] Cisco. "Cisco 7600 Series Router Software Configuration Guide, Cisco IOS Release 15S". 2017. [Online] Available: https://www.cisco.com/c/en/us/td/docs/routers/7600/ios/15S/configuration/guide/7600_15_0s_book/syncE.html.

[16] Dell. "DELL EMC POWERSWITCH S4100-ON". 2020. [Online] Available: https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/dell-emc-networking-S4100-series-spec-sheet.pdf.

[17] Texas Instruments. "IEEE 1588 precision-time protocol (PTP) Ethernet PHY transceiver". 2020. [Online] Available: https://www.ti.com/product/DP83640.

[18] Microsemi. "IEEE 1588-2008 Synchronization PLL". 2020. [Online] Available: https://www.microsemi.com/product-directory/ieee-1588-plls-and-software/4659-zl30347.

[19] ARM. "AMBA: The Standard for On-Chip Communication". 2020. [Online] Available: https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications.

[20] Xilinx. "Vivado Design Suite - HLx Editions". 2020. [Online] Available: https://www.xilinx.com/products/design-tools/vivado.html.

[21] Xilinx. "Vivado User Guide". 2020. [Online] Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug910-vivado-getting-started.pdf.

[22] ARM. "AMBA 4 AXI4-Stream Protocol Specification". 2010. [Online] Available: https://developer.arm.com/documentation/ihi0051/a/.

[23] S. L. Graham, P. B. Kessler, and M. K. Mckusick. "Gprof: A call graph execution profiler". *SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, 1982.

[24] ARM. "ARM Map". 2020. [Online] Available: https://www.arm.com/products/development-tools/server-and-hpc/forge/map.

[25] Intel. "Intel VTune Profiler". [Online] Available: https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler/analyze-and-filter-data.html.

[26] Intel. "Intel VTune Profiler". [Online] Available: https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[27] AMD. "CodeXL Quick Start Guide". 2012. [Online] Available: http://developer.amd.com/wordpress/media/2012/10/CodeXL_Quick_Start_Guide.pdf.

[28] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard". *Parallel Computing*, pages 22(6):789–828, 1996.

[29] "DataDog". 2020. [Online] Available: https://docs.datadoghq.com/watchdog/.

[30] The MPI Forum. "MPI: A message passing interface". *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 878–883, 1993.

[31] Argonne National Laboratory. "Performance Visualization for Parallel Programs". [Online] Available: https://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm.

[32] Jason G Tong. "Software profiling for an FPGA-based CPU core". pages 38–44, 2007.

[33] Intel. "SignalTap User's Guide". 1999. [Online] Available: https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/ug/signal.pdf.

[34] Xilinx. "Intergrate Logic Analyzer". 2016. [Online] Available: https://www.xilinx.com/products/intellectual-property/ila.html.

[35] Xilinx. "AXI Performance Monitor v5.0: LogiCORE IP Product Guide". 2017. [Online] Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_perf_mon/v5_0/pg037_axi_perf_mon.pdf.

[36] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. "PAPI: A Portable Interface to Hardware Performance Counters". *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[37] Leslie Shannon and Paul Chow. "Using reconfigurability to achieve real-time profiling for hardware/-software codesign". *FPGA '04 Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 190–199, 2004.

[38] Ioannis Parnassos, Panagiotis Skrimponis, Georgios Zindros, and Nikolaos Bellas. "SoCLog: A Real-Time, Automatically Generated Logging and Profiling Mechanism for FPGA-based Systems On Chip". *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1946–1488, 2016.

[39] Daniel Nunes, Manuel Saldana, and Paul Chow. "A profiler for a heterogeneous multi-core multi-FPGA system". *2008 International Conference on Field-Programmable Technology*, 2008.

[40] Manuel Saldana, Daniel Nunes, Emanuel Ramalho, and Paul Chow. "Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI". *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 2006.

[41] Paul Emmerich, Sebastian Gallenmuller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "MoonGen: A Scriptable High-Speed Packet Generator". *2015 Internet Measurement Conference (IMC)*, pages 275–287, 2015.

[42] DPDK. "Data Plane Development Kit". 2019. [Online] Available: https://www.dpdk.org.

[43] Xilinx. "Vivado Design Suite User Guide - High-Level Synthesis". 2018. [Online] Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

[44] Fidus. "Sidewinder 100 Datasheet". 2020. [Online] Available: https://fidus.com/wp-content/uploads/2020/01/Sidewinder_Data_Sheet_2020.pdf.

[45] Xilinx. "Zynq UltraScale+ MPSoC Data Sheet: Overview". 2019. [Online] Available: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[46] Xilinx. "Zynq UltraScale+ MPSoC Data Sheet". 2020. [Online] Available: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[47] ARM. "ARM CORTEX-A53". 2020. [Online] Available: https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a53.

[48] Qianfeng Shen. "100G UDP Link for FPGAs through AXI STREAM (GULF-Stream)". 2019. [Online] Available: https://github.com/UofT-HPRC/GULF-Stream.

[49] Dell Technologies. "DELL EMC Z9100-ON SERIES SWITCHES". 2020. [Online] Available: https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/dell-networking-z9100-spec-sheet.pdf.

[50] Xilinx. "UltraScale+ FPGAs Product Tables and Product Selection Guide". 2015. [Online] Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[51] Alpha Data. "ADM-PCIE-8K5". 2019. [Online] Available: https://www.alpha-data.com/pdfs/adm-pcie-8k5.pdf.

[52] Xilinx. "Zynq 7000 SoC Datasheet Overview". 2018. [Online] Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[53] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow. "Galapagos: A Full Stack Approach to FPGA Integration in the Cloud". *IEEE Micro*, 38(6):18–24, 2018.

Appendix A

Resolution of Latency Measurements

A.1 Bandwidth Adjustment

Section 4.3.1 discussed the resolution of latency measurements in the presence of different amounts of traffic. To adjust the amount of traffic, a hardware module was used to back-pressure on the Traffic Source. Figure A.1 shows the usage of this module in the base latency measurement. This detail was left out of Figure 4.5a for simplicity. The bandwidth Adjuster is a pass-through module that takes in AXI-Stream traffic and outputs it on the next clock cycle. It takes one input, *adjustment_interval*, from the user. The *adjustment_interval* determines how often this module back-pressures on its input traffic. As mentioned in Section 4.3.1, each packet is of size 128 bytes, which is equal to 1024 bits per packet. Equation A.1 calculates how much bandwidth the Traffic Source uses based on the Bandwidth Adjuster interval.

$$BW = \frac{I_{adj} - 1}{I_{adj}} \cdot 1024 \cdot f_{gen} \tag{A.1}$$

where BW is the amount of 100G bandwidth used by the Traffic Source, I_{adj} is the adjustment interval, and f_{gen} is the frequency at which traffic is generated. The factors of 1024b is used to account for the number of transmitted bits per cycle. For example, if the *adjustment_interval* is set to 4 ($I_{adj} = 4$), then the Bandwidth Adjuster back-pressures on its incoming traffic every 4 cycles. In this example, if the incoming traffic is generated at 100 MHz, the 100G bandwidth used by the Traffic source can be calculated using Equation A.2.

$$\begin{aligned} BW &= \frac{3}{4} \cdot 1024b \cdot 100MHz \\ &= 76.8^{Gb}/_{s} \end{aligned} \tag{A.2}$$

A.2 Resolution Measurement

As mentioned in Section 4.3.1, the highest bandwidth usage that did not result in packet drops and latency spikes was found to be 98.304 Gb/s. To find this value, we slowly decreased the adjustment interval from a high number in the experiment of Figure A.1. The highest adjustment interval that did

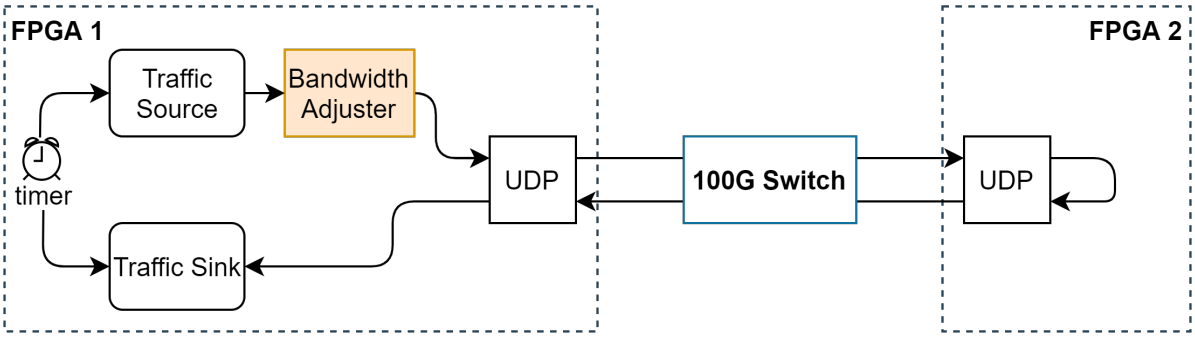

Figure A.1: Bandwidth usage adjustment in latency measurements

not cause packet drops was 25, which results in 98.304 Gb/s.

We continued lowering the adjustment interval from 25 to 2 to characterize the resolution of latency measurements in the presence of different amounts of Traffic. This was shown in Figure 4.8. Table A.1 provides more details about the resolution of latency measurements in the presence of different amounts of traffic.

Table A.1: Resolution of latency measurements in the presence of different amounts of traffic

Adjustment Interval	Bandwidth Usage (Gb/s)	Resolution (micro-seconds)	Resolution (10ns cycles)
25	98.3	6.7	670
24	98.13	3.16	316
23	97.95	2.01	201
22	97.75	1.43	143
21	97.52	1.1	110
20	97.28	0.87	87
19	97.01	0.7	70
18	96.71	0.58	58
17	96.38	0.49	49
16	96	0.41	41
15	95.57	0.35	35
14	95.09	0.3	30
13	94.52	0.26	26
12	93.87	0.22	22
11	93.09	0.19	19
10	92.16	0.16	16
9	91.02	0.13	13
8	89.6	0.11	11
7	87.77	$9 \cdot 10^{-2}$	9
6	85.33	$7 \cdot 10^{-2}$	7
5	81.92	$6 \cdot 10^{-2}$	6
4	76.8	$4 \cdot 10^{-2}$	4
3	68.27	$3 \cdot 10^{-2}$	3
2	51.2	$2 \cdot 10^{-2}$	2

Appendix B

pPTP Interface

Figure B.1 is an example of a connection between a master instance and a slave instance of pPTP in Vivado. Normally, these instances reside in different FPGAs; however, Figure B presents them in a single diagram for simplicity. The connections between the pPTP instance and the timers within pPTP instances are all raw wire connections. The pPTP instances are connected using AXI4-Stream lines. Here, we examine the interface and signals of pPTP master and slave instances in more details.

B.1 pPTP Slave

The slave instance of pPTP initiates the synchronization sequence. To initiate the synchronization, the user must provide the following information to the slave instance:

- **init**: The input register that initializes the synchronization sequence
- **sync_period**: The synchronization interval in number of cycles (the number of cycles between receiving a synchronization response and sending the next synchronization request)
- **remote_ip_tx**: The IP address of the pPTP master; embedded into the AXI4-Stream TUSER field
- **remote_port_tx**: The remote network port used for pPTP communications; embedded into the AXI4-Stream TDEST field
- **local_port_tx**: The local network port used for pPTP communications; embedded into the AXI4-Stream TID field

When the *init* signal is set high, the pPTP slave immediately sends the first synchronization request. The synchronization will continue periodically until *init* is set back to low. The three inputs *remote_ip_tx*, *remote_port_tx*, and *local_port_tx* are used to route pPTP communications through the network. The routing information will be embedded into optional fields of the AXI4-Stream packets. Note that usage of the routing inputs depends on the mode of communication. In the experiments done for this thesis, GULF Stream [48] was used for inter-FPGA UDP communications, in which case the routing information of the pPTP slave was directly sent to GULF Stream. However, if Galapagos [53] is used for communications within the network, only specifying *remote_port_tx* would be necessary.

B.2 pPTP Master

The master instance remains idle until it receives a synchronization request from the slave instance. The synchronization request is the only expected message and any other incoming messages are ignored by the master. The master timer is a free-running clock that starts when the FPGA is programmed and sends its current time to pPTP master at every clock cycle. Upon receiving the synchronization request, master sends a synchronization response to the slave. The payload of the response is the timestamp using the time from the master timer. The pPTP master uses the routing information embedded within AXI4-Stream packets to determine the destination of synchronization responses.

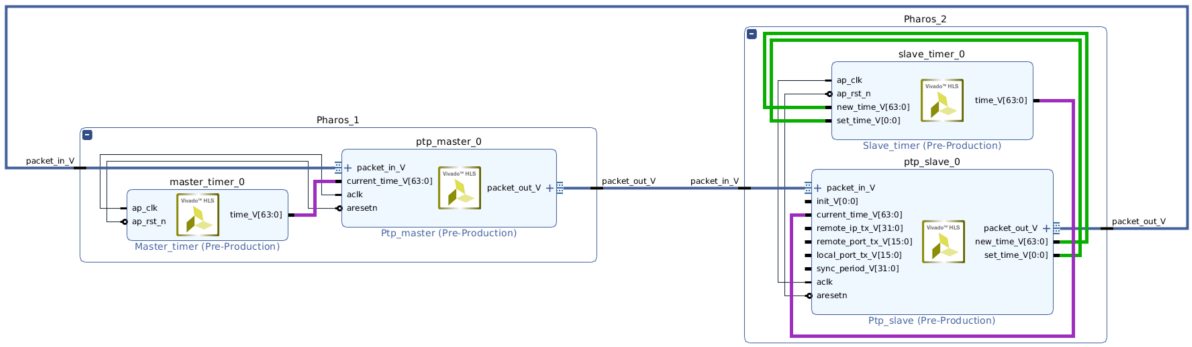

Figure B.1: Connection between master and slave instances of pPTP in Vivado [20]

Appendix C

Resource Utilization

Table C.1 provides details about the hardware utilization of all Pharos components. The Packet Parser is chosen to have a fan-in size of 1000. Section C.1 provides more details about the hardware utilization of the Packet Parser.

Table C.1: Detailed hardware utilization of the Pharos performance monitor

Component	**LUTs**	**FFs**	**BRAMs**
Master Timer	71	65	0
Slave Timer	135	64	0
pPTP Master	201	1	0
pPTP Master	589	195	0
Packet Generator	122	65	0
Packet Parser - 1000	1,122	6,197	5
Packet Snooper	141	131	0
Event Logger	96	5	0
EMA	292	129	0
Total	**2,769**	**6,852**	**5**
Available in Sidewinder	**522,720**	**1,045,440**	**986**

C.1 Packet Parser

As mentioned in Section 3.4.2, the size of the Packet Parser's fan-in, which can be set using a pre-synthesis parameter, represents the maximum number of Packet Generators it can connect to. Table C.2 shows the hardware utilization of the Packet Parser based on the number of Packet Generators it can connect to. The size of the Fan-in is shown in powers of 2. This is because the size increases with the bit-width of the TDEST channel, which is used for network addressing in Pharos.

Table C.2: Relationship between the size of Packet Parser Fan-in and its Resource Utilization

Parser Fan-in Size	**LUTs**	**FFs**	**BRAMs**
32	1,053	1,162	2
64	1,221	1,261	3
128	1,558	1.520	3
256	2,232	2,035	3
512	6,260	3,062	3
1024	6,580	5,111	6
2048	11,636	9,210	11
4096	22,388	17,405	21
8192	43,892	33,792	39
16384	86,900	66,563	74
32768	172,916	132,102	148
65536	344,948	263,177	296