Thanh Bui
Downing College
ttb29

Computer Science Tripos - Part II Project

An accelerated, network-assisted TCP fast retransmit

May 17, 2019

Declaration

I, Thanh Bui of Downing College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Thanh Bui of Downing College, am content for my dissertation to be made available to the students and staff of the University.

SIGNED DATE

Proforma

Name:	**2385F**
College:	**Downing College**
Project Title:	**An accelerated, network-assisted TCP fast retransmit**
Examination:	**Computer Science Tripos — Part II, June 2019**
Word Count:	**11386**[1]
Final Line Count:	**1700**[2]
Project Originator:	Dr Noa Zilberman
Supervisor:	Dr Noa Zilberman

Original Aims of the Project

The aim of this project is to investigate the feasibility and effectiveness of a programmable data plane in application to Transmission Control Protocol (TCP) congestion control. More specifically, I aim to design, implement and evaluate using a programmable switch an implementation that will fast retransmit TCP packet losses that are not due to congestion. The implementation will be assessed through a series of simulations, functional system test and performance analysis. A performance evaluation of the design will also be provided.

Work Completed

The project has been successful. All success criteria were met. I designed an architecture, then implemented, using the P4 programming language and the NetFPGA SUME board, a prototype switch that will fast retransmit TCP packet losses that are not due to congestion. The implementation was simulated and passed both block-level and chip-level simulations, and its functionality was tested in hardware. The switch had a reasonable latency compared to a standard switch and achieved 99.9999% throughput running at full speed of 10Gpbs/port. A performance evaluation of the design was

[1] This word count was computed using `texcount -sum -inc -utf8 -sub=chapter diss.tex` for chapters 1–5.

[2] Approximated using `cat <files> | wc -l` for relevant files.

provided. Two extensions were also completed.

Special Difficulties

The implementation stage of the architecture took longer than anticipated due to limitations of SDNet and the P4→NetFPGA workflow. The current P4→NetFPGA workflow only supports header processing, without deep packet inspection, while this project used programmable buffering logic, which would require the ability to buffer packets. Hence, the design could not be fully expressed in P4 alone. To circumvent this, I had to learn to use Verilog in order to design a different architecture by adding additional HDL modules into the pipeline which allow packet buffering.

Contents

List of figures

Chapter 1

Introduction

In this chapter, I provide the motivation for this project (§1.1) and setup the problem I am solving (§1.2). I also explain some key algorithms involved. Finally, I cover some related work (§1.3).

1.1 Motivation

Transmission Control Protocol (TCP) is the protocol of choice in many data centres. However, it is very sensitive to losses (by design, as a mean for congestion control), which can significantly degrade the performance within the data centres [1]. Congestion control, avoidance and recovery mechanisms are thus of high importance in this field, and a lot of work has been done trying to minimise such loss rate. Still, not all TCP losses are born equal. For example, losses happening at the destination host's network interface card (NIC) are not an indication of congestion within the network. It is assumed that fast retransmission of such lost packets, from within the network, can increase the utilisation of the network.

In-network computing, on the other hand, is an emerging research area in systems and networking, where applications traditionally running on the host are offloaded to the network hardware (e.g. switch, NIC). Examples of applications offloaded in the past include network functions (DNS server [2]), distributed systems functions such as consensus (P4xos [3, 4]), coordination services for modern cloud systems (NetChain [5]) and even a game (Tic-Tac-Toe). Key-Value Store (KVS) caching (NetCache [6]) is also among the popular type of in-network applications. The main promises of in-network computing are performance, both in terms of throughput and latency, and power efficiency [7].

Therefore, in this dissertation, I explore the possibility of applying network-accelerated KVS concepts to TCP fast retransmit mechanism in order to improve cross-datacentre performance. The scope of this project lies at the intersection of computer networking, cloud computing and programmable hardware design. Thus, its success would contribute

towards those areas, especially data centre networks, where low latency and high data rate are desirable.

1.2 Project Aims

A TCP sender normally uses retransmission timeout (RTO)—a simple timer—to recognise and retransmit lost segments. When TCP sends a segment, the timer starts and stops when the acknowledgement is received. If an acknowledgement is not received for a particular segment within a specified time (a function of the estimated round-trip delay time), the sender will assume the segment was lost in the network, and will retransmit the segment.

Fast retransmit is an enhancement to TCP that reduces the time a sender waits before retransmitting a lost segment. Duplicate acknowledgement (DUP ACK) is the basis for the fast retransmit mechanism. After receiving a packet (e.g. with sequence number 1), the receiver sends an acknowledgement by adding 1 to the sequence number (i.e. acknowledgement number 2). This indicates to the sender that the receiver received the packet number 1 and it expects packet number 2. Suppose that two subsequent packets are lost. The next packets the receiver sees are packet numbers 4 and 5. After receiving packet number 4, the receiver sends an acknowledgement, but still only for sequence number 2. When the receiver receives packet number 5, it sends yet another acknowledgement value of 2. DUP ACK occurs when the sender receives more than one acknowledgement with the same sequence number (2 in our example).

DUP ACKs are a sign of an isolated loss. The lack of acknowledgement number progress means 2 hasn't been delivered, but the stream of ACKs means some packets are being delivered (4 and 5 in our example). When a sender receives several DUP ACKs, it can be reasonably confident that the segment with the sequence number specified in the DUP ACK was dropped. A sender with fast retransmit will then retransmit this packet immediately for the retransmission timer to expire.

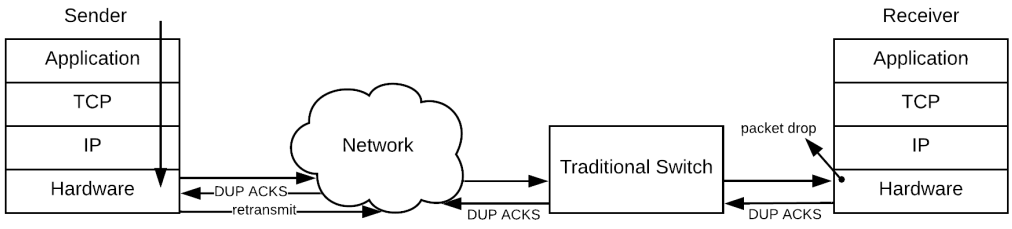


Figure 1.1: The standard convention of TCP fast retransmit.

Currently, the DUP ACKs will traverse all the way back to the sender (Figure 1.1).

The sender receives the DUP ACKs, then retransmits the packet with the next higher sequence number. In a typical data centre network, a packet will traverse multiple hops, so the delay induced by the network and the host is indeed substantial.

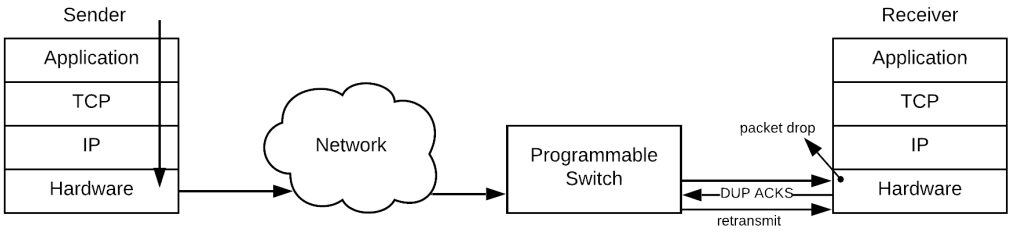


Figure 1.2: The proposed TCP fast retransmit, assisted by the programmable switch.

This project aims to mitigate last-hop packet drops that are not caused by congestion using a programmable switch. The switch will be able to retransmit the packets from within the network, instead of waiting for the DUP ACKs to get back to the host (Figure 1.2), thereby reduces the response time to DUP ACKs and reduces unnecessary changes to the congestion window. The implementation will be based on the KVS concept, where the keys are the flow ID and the packet sequence number, and the value is the payload.

1.3 Related Work

1.3.1 TCP Congestion Control in Data Centres

One of the main aspects of TCP is congestion control [8] where a number of mechanisms are used to achieve high performance and avoid sending more data than the network is capable of forwarding, that is, to avoid causing network congestion. In particular, TCP uses a *congestion avoidance* algorithm that includes various aspects of an additive increase/multiplicative decrease (AIMD) scheme, with other schemes such as *slow start*, *fast retransmit* and *fast recovery* to achieve congestion avoidance. The four intertwined algorithms have been refined over the years and are defined in more detail in RFC 5681 [9].

Together with TCP congestion control, there have been a lot of recent researches focusing on mechanisms to handle congestions in the data centres [10–14] which serve as an inspiration for this project. For instance, Data Center TCP (DCTCP) [11] suggests the use of the Congestion Encountered (CE) code point in the IP header and the ECN-Echo (ECE) flag in the TCP header to allow the sender to compute a congestion estimate and react promptly by reducing the TCP congestion window (cwnd) accordingly.

This project will mostly focus on the *fast retransmit* algorithm, which has been explained in the previous section, and how to assist it using a programmable switch.

1.3.2 Programmable Data Planes

Another key source of inspiration for this project is the emergence of Software-Defined Networking (SDN) [15]. The key idea behind it was to physically decouple the control plane from the data plane, which allows centrally managing the control plane in software, while opening the control logic to the users.

In the past, network devices were fixed-function and supported only the functionality defined by their manufacturer [15]. However, this changes drastically with the introduction of programmable switch-ASICs (Application-Specific Integrated Circuit) [16] and the rise of SmartNICs. Today, the dominant languages used in this field are P4 [17–19] and Protocol Oblivious Forwarding (POF) [20, 21]. They enable faster development/provisioning of new and/or custom protocols, as opposed to the long wait for the release of fixed-function ASIC switches supporting standardised protocols [22].

Different programmable switches have different architectures, but they generally include a programmable pipeline. For example, PISA (Protocol-Independent Switch Architecture) is a single pipeline forwarding architecture (Figure B.1a). The packet is parsed into individual headers. The headers and intermediate results can be used for matching and actions. The headers can also be modified, added or removed. Finally, the packet is deparsed (Figure B.1b). This structure has been used in many applications. One of the most popular applications using a programmable switch is In-band Network Telemetry (INT) [23]. It is a framework designed to allow the collection and reporting of network states, by the data plane, without requiring intervention or work by the control plane. The general idea is that packets will contain header fields that are interpreted as "telemetry instructions" by network devices. These instructions tell an INT-programmable device what state to collect and write into the packet as it transits the network. Examples of collectable states are switch id, ingress port id, ingress timestamp or hop latency.

With more focus, data plane programmability has the potential to benefit future-proof forwarding devices, which are able to support major control plane and protocol updates, without mandating any hardware upgrades.

Chapter 2

Preparation

In this chapter, I first state the software I used (§2.1) and the starting point for this project (§2.2). I move on to present the formal requirements (§2.3). This is followed by a discussion of the different components of a programmable data plane, including the P4 programming language (§2.4), the SDNet compiler (§2.5), the NetFPGA platform (§2.6) and the P4→NetFPGA workflow (§2.7). Finally, I discuss the project workflow (§2.8).

2.1 Software Used

Below I describe and justify, where necessary, the development environment and the programming languages that I used.

2.1.1 Development Environment

- **The NetFPGA SUME**[1] [24] is an open-source platform which provides an accessible development environment that both reuses existing codebases and enables new designs. It uses an advanced, FPGA-based board that features a Xilinx Virtex-7 690T supporting 30 13.1 GHz GTH transceivers. This board easily supports simultaneous wire-speed processing on the four 10Gb/s Ethernet ports, and it can manipulate and process data on-board, or stream it over the 8x Gen3 PCIe interface and the expansion interfaces. It is indeed ideal for any high-performance design such as in this project.
- **P4→NetFPGA** (P4 on NetFPGA) is the workflow to develop and test P4 programs using the Xilinx[2] P4-SDNet[3] toolchain within the NetFPGA SUME reference switch design.
- **Vivado® Design Suite** is a software suite produced by Xilinx for synthesis and implementation of HDL designs. Vivado was used in the project because it is the

[1] A collaborative effort between Digilent, the University of Cambridge and Stanford University.
[2] https://www.xilinx.com/
[3] https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips. Its flexibility also enables me to simulate my design behaviour with different stimuli, synthesize the design to hardware and perform timing analysis.

- **Git** was used for version control, allowing quick roll-back and efficient management of multiple source trees using branches to implement different functionalities at various stages of the project. The repository itself was hosted remotely on GitHub[4]—a free online `git` hosting service.
- **Microsoft OneDrive**[5] was used to back up relevant files throughout the project.

2.1.2 Programming Languages

In this project, I used a multitude of languages, including **P4**, **Verilog**, **Tcl** and **Python**.

- **P4** [17] is a high-level language designed to describe packet processing logic in the packet forwarding planes. Besides, unlike general purpose languages such as C or Python, P4 is domain-specific with a number of constructs optimized around network data forwarding, hence is well-suited for implementing the forwarding plane of network elements such as our switch.
- **Verilog** was used to implement certain HDL modules within the NetFPGA platform, in order to add or modify certain functionalities to suit the purpose of my design. In fact, the NetFPGA platform is mostly Verilog-based, except for the packet-processing pipeline, which is implemented in P4. Thus, this project requires a strong grasp of Verilog.
- **Tcl** was used to write project wrappers and debug scripts because it is part of the Xilinx Vivado development environment.
- **Python** is the language used in the existing test infrastructure since the other languages used are for the hardware level. It was used extensively in the evaluation because of the `scapy` library [25], which enables the user to send, sniff, dissect and forge network packets. This capability allows me to write unit tests for my program by building customised packets, sending and checking them.

I also made use of the `make` build automation tool to automate project builds, tests and benchmarks.

[4] https://github.com/<github-username>/part-ii-proj/
[5] https://onedrive.live.com/about/en-gb/

2.2 Starting Point

This project uses the knowledge about TCP introduced in the Part IB *Computer Networking* course and the experience in Electronic Computer-aided Design (ECAD) and working with a design-flow for Field Programmable Gate Arrays (FPGAs) from Part IB *ECAD and Architecture Practical Classes*.

During the development of this project, I acquired further knowledge from the materials covered in the following courses:

- *High Performance Networking* (Part III P51)—Introduction to P4 and P4→NetFPGA;
- *Principle of Communications* (Part II)—TCP flow control and congestion control. Design choices for scheduling and queue management algorithms for packet forwarding;
- *LaTeX and MATLAB* (Part II)—Typesetting the project proposal and dissertation.

In terms of familiarity, I had no prior experience with P4 and Tcl programming language, the NetFPGA platform and the P4→NetFPGA workflow, and little experience in Verilog from the similar language SystemVerilog learnt in Part IB *ECAD and Architecture Practical Classes*. Therefore, I had to spent some time learning the languages and the workflow. I had some prior experience in Python and Git from various projects and internships.

The NetFPGA platform provides an infrastructure for the project, and some reference codes. I used from this infrastructure the existing interfaces, DMA, and most of the HDL modules and externs, modifying some of them where appropriate, and wrote my own P4 code for the core functionalities of the design. This means that all the P4 code and the Python tests were written from scratch, while the code for the additional HDL modules and externs in Verilog, as well as the project wrappers in Tcl, are modified to suit the required functionalities from some of the current modules in the existing infrastructure.

Codes modified during the project contain a NetFPGA license which allows any modification and usage that is in compliance with the license. Any open-source figure used in this dissertation also has a permissive license that allows such usage, and will be acknowledged accordingly.

2.3 Requirements Analysis

This project has one software deliverable—an implementation for a programmable switch that will retransmit a packet when it receives the third DUP ACK from the

receiver—which includes codes for the following: data plane, control plane, simulation environment and test environment.

Below is a list of requirements and extensions for the deliverable, refined from the success criteria described in the original proposal (Appendix D) and prioritised using *MoSCoW* criteria [26]:

Must have

- Have an implementation of the switch's functionalities in P4 code.
- The implementation passes an SDNet simulation (block-level simulation).
- The implementation passes a SUME simulation (chip-level simulation).
- The implementation works correctly in hardware (functional system test).
- A demonstration of the switch's interoperability with a software-based application.
- A performance evaluation of the design.

Should have

- The switch will send a notification to the source if the retransmission fails.
- A performance evaluation in comparison to a standard switch performance.

Could have

- The design will support different packet sizes.
- The design will support more than a single flow, and support the configuration of flows to monitor.
- The design has the ability to adaptively add or remove flows to monitor.

Won't have

- The implementation will not be simulated using network simulators such as ns2 or omnet++.

2.4 The P4 Language[6]

P4 (Programming Protocol-independent Packet Processors) has become the *de facto* standard language for describing how network packets should be processed, and is becoming widely used by many developers in conjunction with SDN control planes. This section gives a brief overview of the P4 programming language with the aim to provide sufficient basis to understand the project.

[6]This explanatory section is based on some contents from [17, 18, 27].

Most targets, if not all, implement both a control plane and a data plane. P4 is designed to specify only the data plane functionality of the target. P4 programs also partially define the interface by which the control plane and the data plane communicate, but P4 itself cannot be used to describe the control plane functionality of the target. Thus, in the remaining of this dissertation, when we refer to P4 as "programming a target", we mean "programming the data plane of a target". Figure 2.1 describes the canonical process of programming a P4 target. The vendor of a packet processing device provides three components to the user:

- The packet processing target device.
- A P4 architecture model to expose the programmable features of the target to the programmer.
- A compiler to map the user's P4 program into a target-specific configuration binary file which is used to tell the target how it should be configured to process packets.

The programmer will write a P4 program to instantiate the architecture model, by filling its programmable components. The programmer also provides control software (i.e. a control plane) which is responsible for controlling the packet processing device at run time.

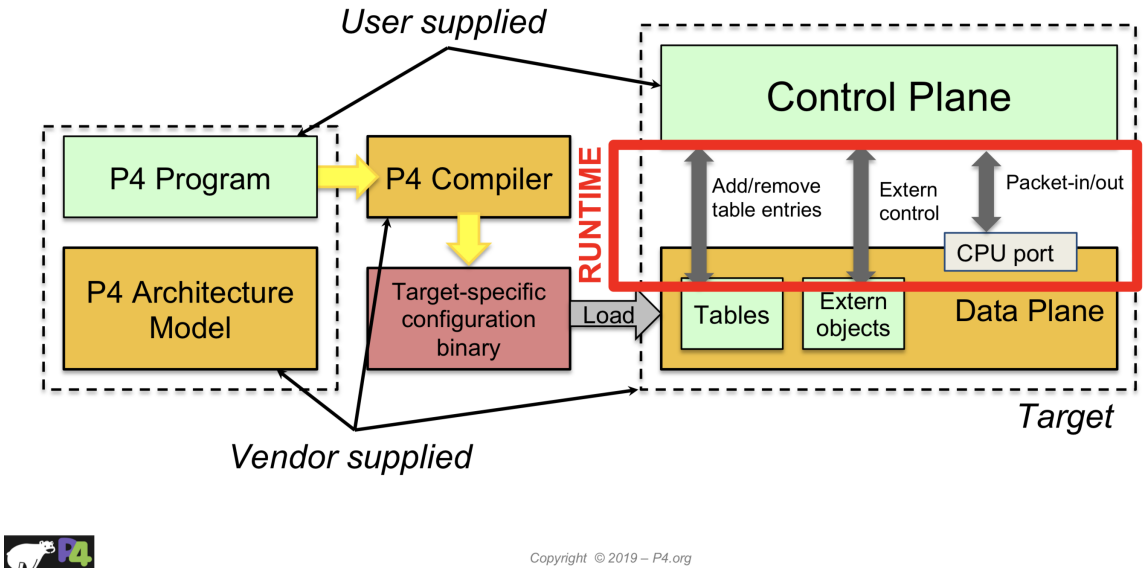


Figure 2.1: The process of programming a P4 target. Source: P4.org – Copyright © 2019.

In order to make the devices "protocol-independent", i.e. without built-in implementations of specific protocols, P4 allows us to define the format of all protocol headers that we want the device to handle using the `header` keyword. Here is an example that shows the definition of the Ethernet header in the project. The IPv4 and TCP headers are defined similarly. Note that `typedef` statements can also be used to make the code more readable.

```
typedef bit <48> EthAddr_t;

header Ethernet_h {
  EthAddr_t dstAddr;
  EthAddr_t srcAddr;
  bit<16> etherType;
}

header IPv4_h {
  bit<4> version;
  ...
}

header TCP_h {
  bit<16> srcPort;
  ...
}

struct Parsed_packet {
  Ethernet_h ethernet;
  IPv4_h ip;
  TCP_h tcp;
}
```

This makes P4-programmable switch differ from a traditional switch in two fundamental ways:

- The data plane functionality is defined by the P4 programmer, rather than by the manufacturer of the switch. The data plane is configured at initialisation time to implement the functionality described by the P4 program and has no built-in knowledge of existing network protocols.
- The set of tables and other objects in the data plane are no longer fixed, but defined by the P4 program. The P4 compiler then generates the API that the control plane uses to communicate with the data plane, using the same channels as in a fixed-function device.

In this project, we will be using the SDNet compiler ("the compiler"), the NetFPGA SUME board ("the target device"), and the SimpleSumeSwitch architecture of the P4→NetFPGA workflow ("the architecture model"), all of which will be described in more detail in the next three sections.

2.5 The Xilinx SDNet[7]

The Xilinx SDNet compiler is the centerpiece of the P4→NetFPGA workflow. It is the Xilinx SDNet original design environment for an internally-created packet processing language called PX [29], with a P4 to PX translator. Figure 2.2 depicts the process of compiling P4 programs that target the SimpleSumeSwitch architecture using SDNet. The front end translator maps P4 programs into corresponding PX programs and also

[7]This explanatory section is based on some contents from [27, 28].

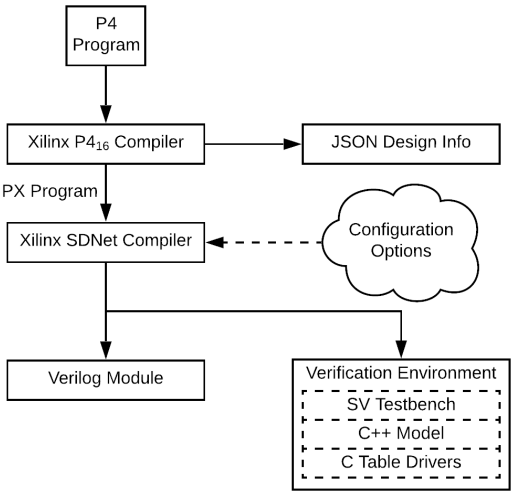


Figure 2.2: The Xilinx SDNet compilation flow. P4 programs are first translated into a PX program, which is then compiled into a Verilog module using the SDNet flow. SDNet also produces a verification environment.

produces a JSON file with information about the design that is required by the runtime control software. The PX program is passed, along with configuration parameters, into SDNet which then produces an HDL module that implements the user's P4 program, and has standard AXI-Stream packet interfaces and an AXI-Lite control interface. SDNet generated designs can be configured to process packets at line rates between 1 and 400 Gb/s, hence is able to easily handle the aggregate 40G rate in the SUME reference switch design. SDNet also produces a SystemVerilog simulation testbench, C drivers to configure the PX tables, and an optional C++ model of the PX program to be used for debugging purposes.

2.6 The NetFPGA Platform[8]

The NetFPGA (Networked FPGA) project is a teaching and research tool designed to allow packets to be processed at line-rate in programmable hardware. It consists of four components: boards, tools and reference designs, a community of developers and contributed projects. The SUME board that was used in this project, which has I/O capabilities for 100 Gb/s operation such as NIC, multiport switch, firewall, or test/measurement environment, is the latest product in the NetFPGA hardware family.

Figure 2.3 depicts a block diagram of the canonical NetFPGA reference design which is used for switches, NICs, and IPv4 routers. It consists of four 10G SFP+ input/output ports along with one DMA interface for the CPU path. The NetFPGA data path consists of three main components: Input Arbiter, Output Port Lookup, and Output

[8]This explanatory section is based on some contents from [24, 27].

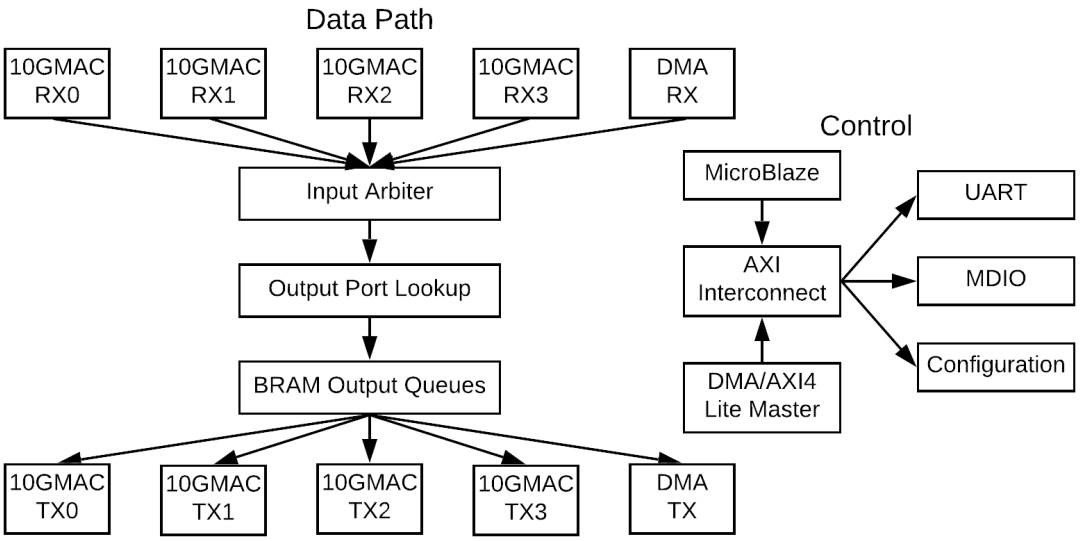


Figure 2.3: Block diagram of the NetFPGA reference switch design.

Queues. The Input Arbiter admits packets from the ports into the data path, towards the Output Port Lookup Module, where the main packet processing occurs and an output port is selected. The Output Queues buffer packets while they wait to be sent to the outputs. The core data path uses a 256-bit wide bus and runs sufficiently fast at 200 MHz to support an aggregate of 40 Gb/s from all four SFP+ ports.

The limitation of this platform is that it requires a substantial knowledge in both hardware design and networking, with programs written in Verilog or VHDL. To overcome this, the P4→NetFPGA workflow was created to make it much easier to process packets in hardware and prototype new systems without being bogged down in hardware development.

2.7 The P4→NetFPGA Workflow[9]

Figure 2.4 outlines the automated P4→NetFPGA workflow [27]. We first write a P4 program which is compiled (by Xilinx P4-SDNet) into an HDL instance of the *SimpleSumeSwitch* architecture. The SimpleSumeSwitch module is then automatically integrated into the NetFPGA reference switch design by replacing the default Output Port Lookup module. The SimpleSumeSwitch module is the core of the pipeline as it defines the logic of the programmable device. There have been important works that have taken this same structure [3, 30].

[9]This explanatory section is based on some contents from [27, 28].

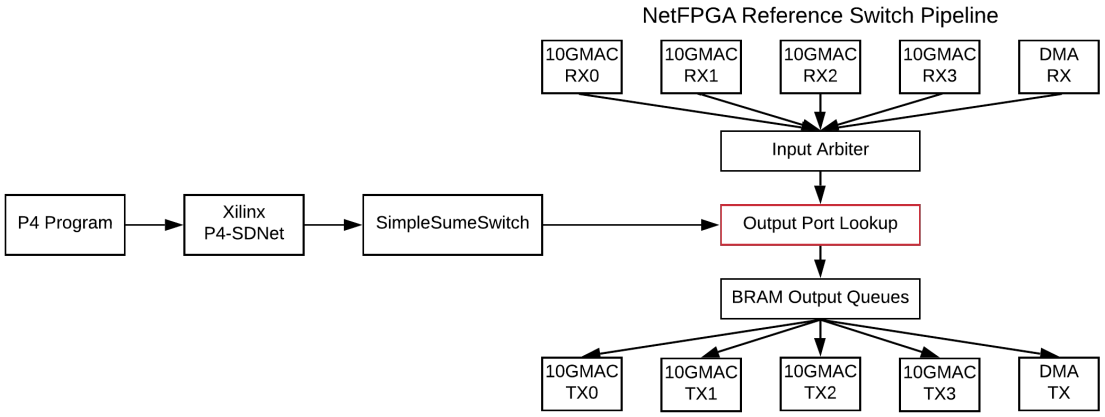


Figure 2.4: The automated P4→NetFPGA compilation flow. P4 programs are compiled into an HDL instance of the SimpleSumeSwitch architecture, which is then used to replace the Output Port Lookup module in the NetFPGA Reference Switch Design.

The SimpleSumeSwitch is the P4 architecture that is currently defined for the NetFPGA SUME board. The architecture consists of a single parser, a single match-action pipeline, and a single deparser, as shown in Figure 2.5. In addition, the architecture also defines the format of any standard metadata buses. Standard metadata buses, conveying sideband data alongside each packet, are used to allow P4-programmable elements to interact with non-programmable elements within the architecture. The SimpleSumeSwitch's `sume_metadata` bus is defined as follows:

```
struct sume_metadata_t {
  bit<16> dma_q_size;     // measured in 32-byte words
  bit<16> nf3_q_size;     // measured in 32-byte words
  bit<16> nf2_q_size;     // measured in 32-byte words
  bit<16> nf1_q_size;     // measured in 32-byte words
  bit<16> nf0_q_size;     // measured in 32-byte words
  bit<8> send_dig_to_cpu; // send digest_data to CPU
  port_t dst_port;        // one-hot encoded (see below)
  port_t src_port;        // one-hot encoded (see below)
  bit<16> pkt_len;        // (bytes) unsigned int
}
```

where the format of the `dst_port` and `src_port` fields is:

```
  bit-7     bit-6     bit-5     bit-4     bit-3     bit-2     bit-1     bit-0
(nf3_dma)-(nf3_phy)-(nf2_dma)-(nf2_phy)-(nf1_dma)-(nf1_phy)-(nf0_dma)-(nf0_phy)
```

and the functionality of each field is described by Table 2.1.

The formats of the `digest_data` and the `user_metadata` are also defined by the P4 programmer. They can be used to pass any additional information from the parser to the M/A pipeline and from the M/A pipeline to the deparser. The in/out `control` signals are used to add/removed entries from tables and read/write control registers.

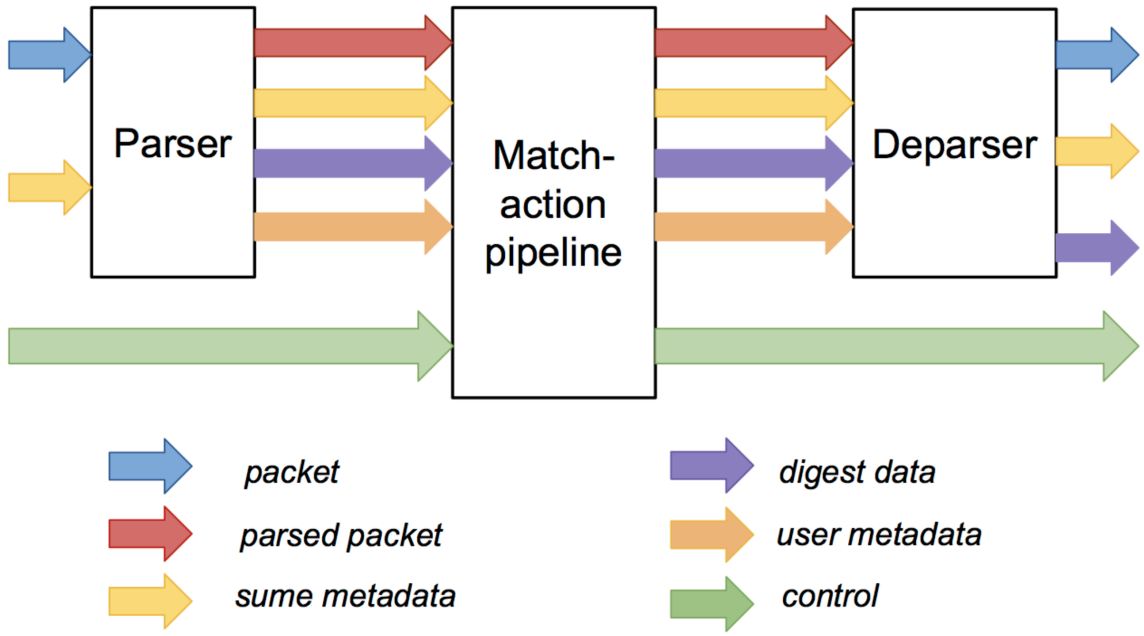


Figure 2.5: Block diagram of the SimpleSumeSwitch P4 architecture used within the P4→NetFPGA workflow. Source: P4→NetFPGA Home Wiki.

Table 2.1: Description of the SimpleSumeSwitch `sume_metadata` fields.

Field Name	Size (bits)	Description
`pkt_len`	16	Size of the packet in bytes (not including the Ethernet preamble or FCS)
`src_port`	8	Port on which the packet arrived (one-hot encoded)
`dst_port`	8	Set by the P4 program – which port(s) the packet should be sent out of (one-hot encoded)
`send_dig_to_cpu`	8	Set the least significant bit of this field to send the `digest_data` to the CPU
`*_q_size`	16	Size of each output queue at P4 processing start time, measured in 32-byte words

The SimpleSumeSwitch is a good architecture because it is simple and easy to understand, yet remains flexible enough to allow developers to implement a variety of different networking protocols and algorithms. Its flexibility also means that it could be extended or completely replaced by writing a new architectural model. For this project, I will modify the NetFPGA Reference Switch Pipeline to include a Cache Queue that will buffer packets. The customised architecture will be explained in more detail in §3.1.

To sum up, the P4→NetFPGA workflow includes the following steps:

1. Write P4 program.
2. Implement custom extern modules, if any.
3. Write Python scripts to generate test data for SDNet simulations.
4. Run HDL simulations.

5. Build bitstream for FPGA.
6. Test the design on hardware.

2.8 Project Workflow

2.8.1 Preparation Stage

The preparation stage happened in the first three weeks of the project. I spent the first week revisiting the basics of TCP and learning in depth its fast retransmit and recovery mechanisms. In the next two weeks, I learned the P4 language, set up the development environment and learned the P4→NetFPGA workflow.

The P4 Language Consortium [19] provides a set of exercises to get me started. I completed their tutorial[10], from which I learned the language basics such as basic forwarding and basic tunnelling. I also learned to use P4 tables and actions to implement advanced behaviour such as source routing and load balancing.

Most of the time spent in setting up the NetFPGA environment went into getting approval for access to the live development repositories, including the P4→NetFPGA and the NetFPGA-SUME codebase, and various licenses and tools necessary to use the P4→NetFPGA toolchain (Xilinx P4-SDNet and Vivado Design Suite). Where appropriate, the licenses are quoted at the beginning of the file. The NetFPGA SUME board was already installed and configured, and is connected to a machine with the appropriate system requirements and dependencies located in the Computer Laboratory. To access the board from my personal device, I set up a Virtual Private Network (VPN) and `ssh` to the machine.

Finally, I learned the P4→NetFPGA workflow through a series of exercises, provided by the NetFPGA Github Organisation[11]. The workflow provides a template for a general P4 program following the SimpleSumeSwitch architecture model, from which I will start to write my implementation, as stated in §2.2. The main challenge of this part is learning Verilog and Tcl in a short amount of time.

2.8.2 Implementation Stage

Following the preparation stage, the implementation stage of this project will take an iterative approach, as illustrated by the workflow in Figure 2.6. After laying out the requirements and designing the architecture, I will start to write the implementation in

[10] https://github.com/p4lang/tutorials
[11] https://github.com/NetFPGA

P4 and test it by running an SDNet simulation, which is written in Python. Then, I will code the HDL modules and configure the system, which is followed by a SUME simulation. The next step will then be compiling the entire design into bitstream for FPGA programming and testing it in hardware, including a static timing analysis. All the steps are iterative: a code review is conducted after each "Coding" step and the outcome of each simulation step provides feedback for refining and improving the design in the next iteration. Finally, when the implementation passes all the tests, I will begin to evaluate its performance.

In order to fulfil the requirements analysis, I follow the *spiral development model* [31] with an iteration count equal to the number of major functionalities to add. This allows for continual implementation, testing and integration of the different functionalities.

2.8.3 Risk Analysis

The P4→NetFPGA workflow is a complex platform that required the knowledge of a multitude of languages, with limited documentation [32] and community support [33]. A potential risk for the project was the difficulty of being sufficiently proficient with the platform to modify its core components and hence the inability to implement the design. Complete failure to do so was unlikely, but it could have consumed a significant amount of development time. As suggested by the spiral development model [31], this high-risk part was scheduled early and some "catch-up" time was allocated in the project timetable in case it caused significant delays.

I have also checked with my supervisor for potential ethical aspects in this project, and this work does not raise any ethical issues.

2.8.4 Backup Plan

Throughout the project development, I made sure to follow good backup procedure by keeping local daily backups of my project using Time Machine for macOS. This provides recent history through incremental backups. I ensured additional remote storage by backing up with Microsoft OneDrive and Git, which also provided version control.

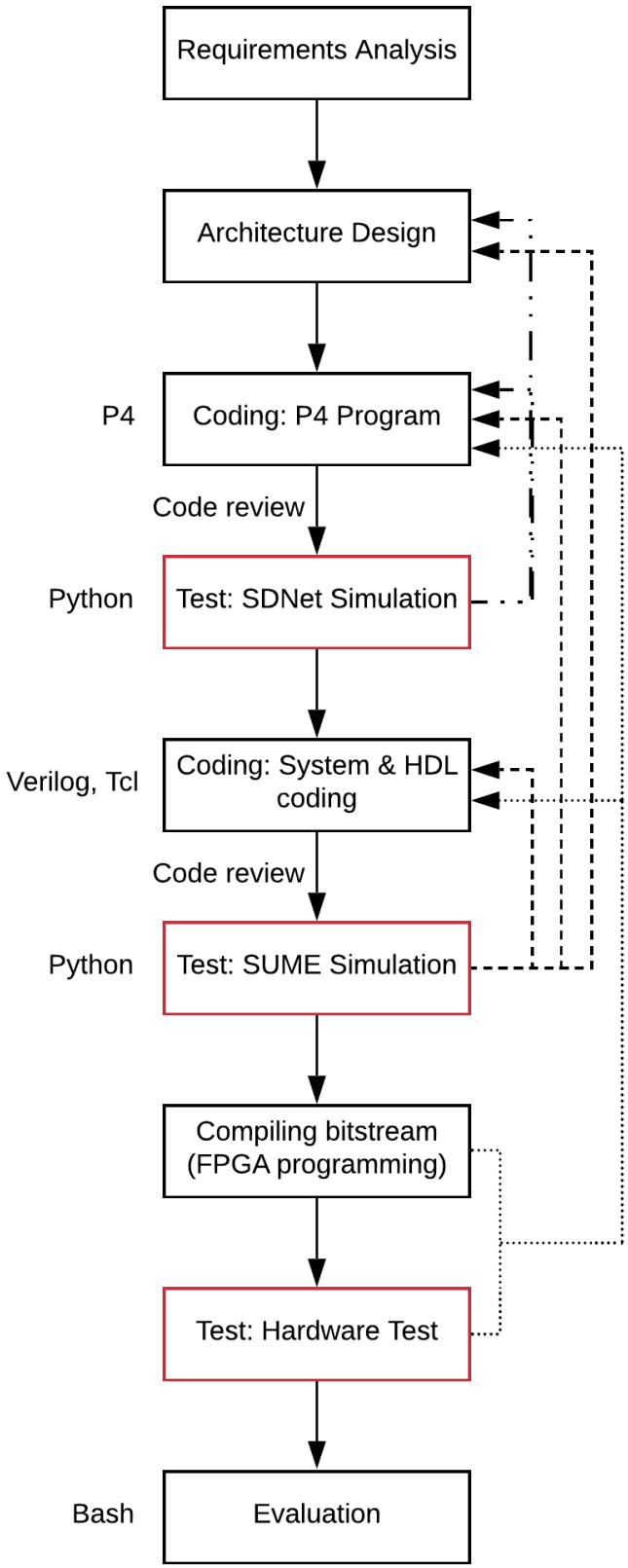


Figure 2.6: Block diagram showing the workflow of the implementation stage. Dotted arrows represent a revision of previous steps, possibly with adjustments/refinements, in an iterative approach. Where appropriate, the programming language involved is stated. Passing all the steps in red box indicates the design meeting the requirements.

Chapter 3

Implementation

This chapter discusses in detail the design and implementation of the functionalities of the programmable switch. First, I describe different design architectures and why a particular architecture is chosen (§3.1). Then, I move on to explain the P4 implementation of the core logic of the switch (§3.2). Lastly, I discuss the additional Verilog components needed to implement the design in hardware (§3.3). A repository overview is also given in Appendix A.

3.1 System Architecture

Following the preparation stage, the first objective is to design the system architecture for the switch. The P4 program will then use the architecture as a reference to implement the core logic, making adjustments to the design where necessary.

3.1.1 Network Level

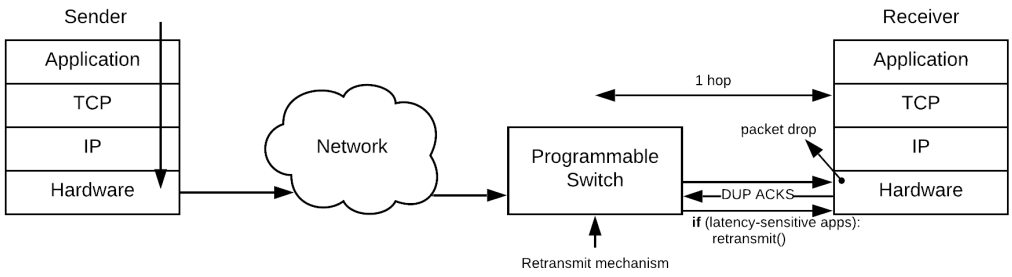


Figure 3.1: The network-level view of the programmable switch. It will be located at the last hop before the receiver, and only performs the fast retransmit on packets from latency-sensitive applications, which are identified by their flow identifier.

To mitigate last-hop drops that are not caused by congestion, the switch will embed the TCP fast retransmit mechanism and will be located in the Top of Rack (ToR) switch (one hop away from the receiver) and not the spine switch (in the core of the network),

as illustrated in Figure 3.1. Since the switch is programmable, it will be configured to buffer only packets from latency-sensitive high priority applications, identified by their unique **flow identifier**, while transmitting other packets normally. This design avoids introducing unnecessary buffering of packets of other flows which is the cause of bufferbloat.

3.1.2 System Level

The initial design followed the NetFPGA reference switch design architecture (Figure 2.3), in which the SimpleSumeSwitch module will embed the buffering logic of the switch. When a packet arrives, the switch computes its unique flow identifier from the 5-tuple (IP address pair, Layer 4 port pair and the protocol, which is TCP in this project). If the flow identifier matches the set of pre-defined flows of interest, it will buffer the packet to retransmit. We use a hash table to map flow identifier to another hash table with the packet sequence number as the key and the packet itself as the value. If the packet is an ACK packet, the switch checks if it is a new ACK and updates relevant counters. If it is a DUP ACK, it increments the ACK counter and retransmits if 3 DUP ACKs are received. Figure 3.2 presents the flowchart showing the buffering logic of this design, with the steps that involve a hash table highlighted in red box.

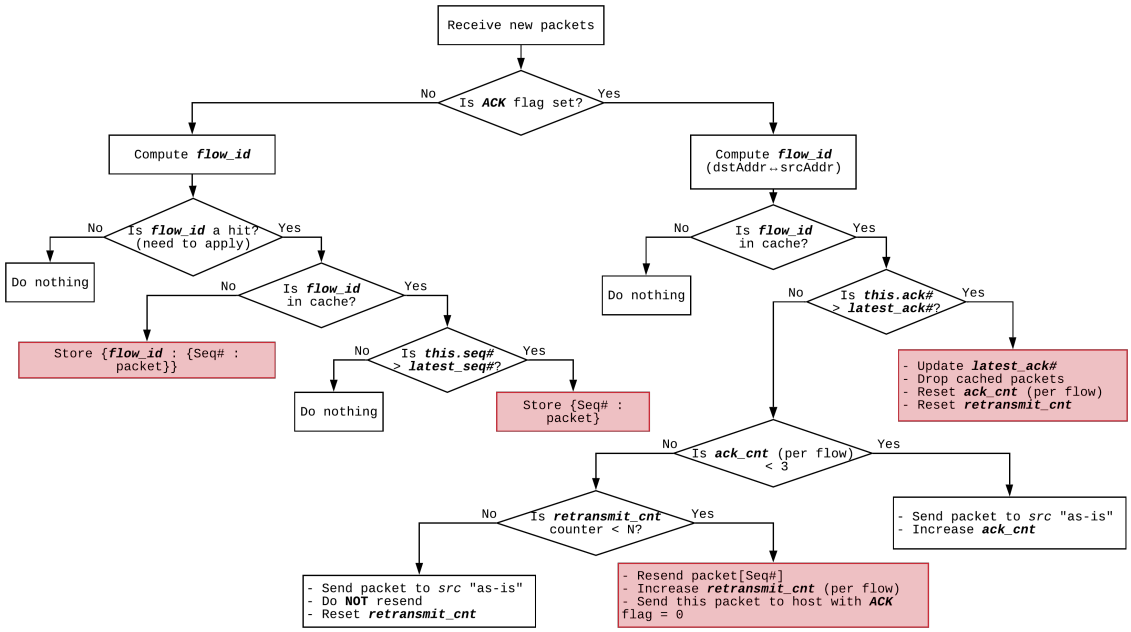


Figure 3.2: Flowchart of the packet retransmit logic. The steps in red box require the ability to store the packet payload.

However, this design did not work due to the limitations of SDNet and the SimpleSumeSwitch architecture of the NetFPGA platform. The SimpleSumeSwitch architecture only supports programmable packet processing without deep packet inspection, i.e.

operations on packet headers only. This means that the P4 program would not be able to access the packet payload to store it in the second hash table. Hence, this design cannot be fully expressed in P4 alone.

After careful study of the initial design and the limitations of the platform, I improved my design by adding an additional HDL module that follows the SimpleSumeSwitch module in the original NetFPGA reference switch pipeline (Figure 2.3). The additional module is called the *Cache Queue* and it will buffer packets to be retransmitted. The buffering logic remains largely similar to what was discuss previously. This addresses the issue of P4 programs being unable to access the packet payload. Our reference switch pipeline will now look like Figure 3.3. In this new pipeline, when a packet exits the SimpleSumeSwitch module, it will be duplicated and buffered in both the output queue and the cache queue. While the output queue sends the packet to the output ports as soon as it can, the cache queue will hold on to the packet. Once there is a "signal" from another packet, the cache queue will drop or send the packet to the output accordingly. The role of the Output Arbiter is similar to that of the Input Arbiter—merging multiple input streams into one output stream—albeit having different names. Section 3.3 will explain the implementations of both the Cache Queue and the Output Arbiter in more detail.

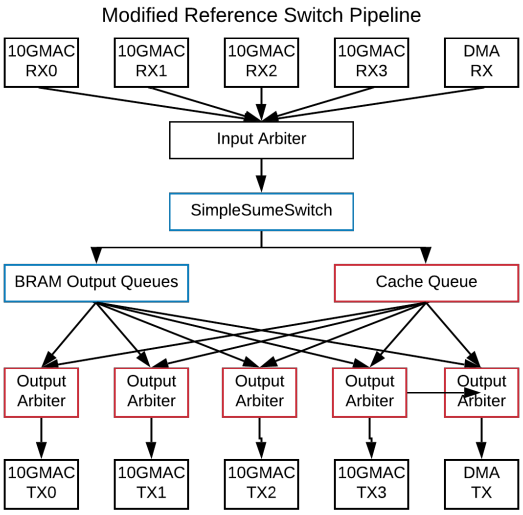


Figure 3.3: Block diagram of the modified reference switch pipeline. Packets are duplicated after the SimpleSumeSwitch module and being buffered in the Cache Queue. Red blocks represent additional modules. Blue blocks represent modules from the reference switch design that are modified.

3.2 P4 Implementation

As previous introduced in §2.7, our P4 program follows the SimpleSumeSwitch architecture which consists of a parser, a match-action pipeline, and a deparser, shown in Figure 2.5.

3.2.1 The Parser

Parsers are functions that are responsible for extracting headers out of an incoming packet, written in a state machine style [18]. We can declare a parser with the following code sequence:

```
// Parser Implementation
@Xilinx_MaxPacketRegion(1024)
parser TopParser(packet_in b,
                 out Parsed_packet p,
                 out user_metadata_t user_metadata,
                 out digest_data_t digest_data,
                 inout sume_metadata_t sume_metadata);
```

where `@Xilinx_MaxPacketRegion` is Xilinx P4-SDNet's additional annotation for parser/deparser that declares the largest packet size (in bits) the parser/deparser needs to support.

Figure 3.4 illustrates the general structure of a parser state machine, which includes three predefined states:

- `start` – the start state.
- `accept` – indicating successful parsing.
- `reject` – indicating a parsing failure.

and other internal states that may be defined by the user. Parsers always start in the `start` state, execute one or more statements, then make a transition to the next state until reaching either the `accept` or `reject` states, which are distinct from the user-defined states and are logically outside of the parser.

An architecture must specify the behaviour when the `accept` and `reject` states are reached. For example, an architecture may specify that all packets reaching the `reject` state are dropped without further processing. Alternatively, it may specify that such packets are passed to the next block after the parser, with intrinsic metadata indicating that the parser reached the `reject` state, along with the error recorded. The SimpleSumeSwitch architecture users the SDNet which does not support `reject`. Hence, the `reject` state is manually defined and implemented to drop all packets without further processing.

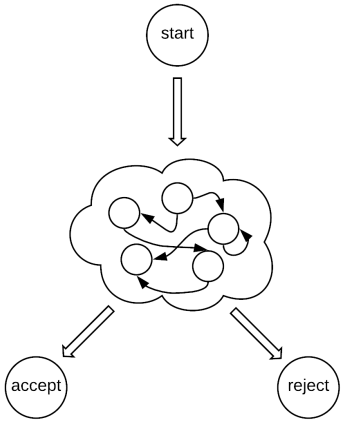


Figure 3.4: The general state machine structure of a parser.

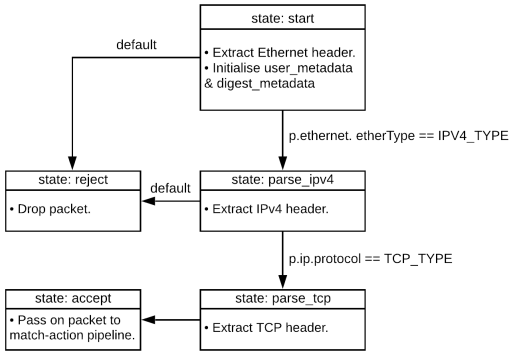


Figure 3.5: The state machine of the design.

Figure 3.5 describes the state machine structure of our parser, which includes a `start` state (Figure 3.6) and two additional user-defined states `parse_ipv4` (Figure 3.7) and `parse_tcp` (Figure 3.8). The P4 `select` statement is used to branch in a parser. It is similar to `case` statement in C or Java, but without "fall-through behaviour"—i.e., `break` statements are not needed. Here, our parser first uses the `packet_in` object's `extract` method to fill out the fields of the Ethernet header. It also initialises the values of the `user_metadata`'s field `digest_data`'s fields to 0. It then transitions to either the `parse_ipv4` state or the `reject` state based on the value of the Ethernet header's `etherType` field. In the `parse_ipv4` state, the parser extracts the packet's IPv4 header, looks at its `protocol` field and transitions to the `parse_tcp` state only if it is `TCP_TYPE` which is defined to be 6. Otherwise, the packet is rejected. Finally, in the `parse_tcp` state, the parser simply extracts the TCP header and then transitions to the `accept` state, where the packet will be passed to the match-action pipeline. A `parse_ethernet` state could be defined similarly to `parse_ipv4` and `parse_tcp`, but I decided to include the parsing of the Ethernet header within the `start` state, together with initialising the metadata, for simplicity.

```
state start {
  b.extract(p.ethernet);
  user_metadata.unused = 0;
  digest_data.unused = 0;
  digest_data.flow_id = 0;
  digest_data.tuser = 0;
  transition select(p.ethernet.etherType) {
    IPV4_TYPE: parse_ipv4;
    default: reject;
  }
}
```

Figure 3.6: The definition of `start` state.

```
state parse_ipv4 {
  b.extract(p.ip);
  transition select(p.ip.protocol) {
    TCP_TYPE: parse_tcp;
    default: reject;
  }
}
```

Figure 3.7: The definition of `parse_ipv4` state.

```
state parse_tcp {
  b.extract(p.tcp);
  transition accept;
}
```

Figure 3.8: The definition of `parse_tcp` state.

3.2.2 The Match-Action Pipeline

A match-action pipeline is a control block where the match-action packet processing logic is implemented. A match-action pipeline uses *tables*, *actions*, and *imperative code* (indicated by the **`control`** keyword) to manipulate input headers and metadata [27]. This match-action processing model was originally introduced as the core around which the OpenFlow model for SDN was built [34]. Our match-action pipeline can be defined by the following code sequence:

```
// Match-action pipeline
control TopPipe(inout Parsed_packet p,
                inout sume_metadata_t sume_metadata,
                inout digest_data_t digest_data,
                inout user_metadata_t user_metadata) {
  /** actions **/
  /** tables **/
  /** logic **/
}
```

This pipeline receives four inputs: the parsed packet `p`, the SUME metadata, the digest data and the user metadata. The direction `inout` indicates that the parameters are both an input and an output. Thus, their values, including the fields in the headers of packet `p`, can be modified. Nonetheless, the `user_metadata` field was not used in this design.

When we defines a match-action table (using the P4 **`table`** keyword), we declare various properties such as the header and/or metadata field(s) to match upon, the type of match to be performed, a list of all possible actions that can be invoked, the number of entries to allocate for the table, and a default action to invoke if no match is found. A table entry contains a specific key to match on, a *single* action to invoke when the entry produces a match, and any data to provide to the action when it is invoked. The table entries are populated at runtime by the control plane software.

Figure 3.9 illustrates the control flow program acting on a packet going through our match-action pipeline, which comprises two match-action tables: `forward` and `retransmit`. The `forward` table (Table 3.1) uses the Ethernet destination address to determine the output port for the next hop. For instance, if the destination

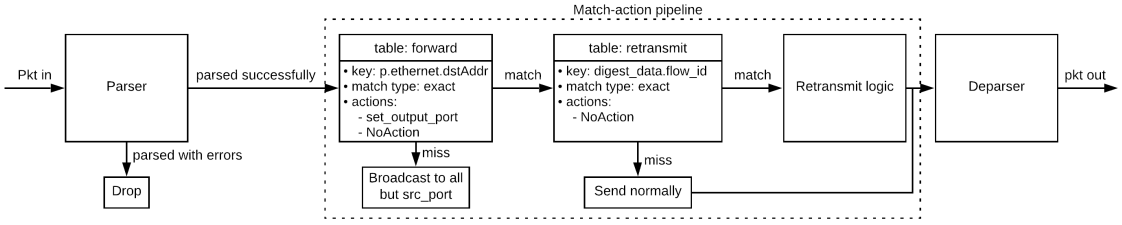


Figure 3.9: The packet processing program of the switch. Incoming packets will go through a parser, a match-action pipeline, which including the buffereing logic and a deparser, before coming out to the output queue and the cache queue.

Table 3.1: Entries for the `forward` table.

Key	Action	Action Data
08:11:11:11:11:08	`set_output_port`	0b00000001
08:22:22:22:22:08	`set_output_port`	0b00000100
08:33:33:33:33:08	`set_output_port`	0b00010000
08:44:44:44:44:08	`set_output_port`	0b01000000
ff:ff:ff:ff:ff:ff	`set_output_port`	0b01010101

Table 3.2: Entries for the `retransmit` table.

Key	Action	Action Data
792281630049477301766976897099	NoAction	_

MAC address is `08:11:11:11:11:08`, the output port will be set to `nf0` (the first port). If the destination address is `08:22:22:22:22:08`, the output port will be set to the second port `nf1`, and so on. If this lookup fails, the packet is broadcasted to all ports except for the source port by xor-ing the source port with `0b01010101`: `sume_metadata.dst_port = 0b01010101^sume_metadata.src_port`.

The `retransmit` table (Table 3.2) checks the computed flow identifier of the packet, which is stored in the `digest_data.flow_id` field: if it matches our flow of interest, the packet will be monitored to assist the TCP fast retransmit process. Otherwise, the packet will just be delivered normally. This allows us to specify the set of flows from latency-sensitive applications by adding their unique flow identifier to the table, in decimal representation. The key in Table 3.2 represents—in decimal—the 104-bit flow identifier of a flow from IP address `10.0.0.1` port `55` to IP address `10.0.0.2` port `75`. Since the retransmit logic is complicated and cannot be expressed using a single action, no action is taken if there is a match, hence no action data is provided. Instead, the packet will be inspected more carefully through a series of actions.

In our match-action pipeline, the following actions were defined: `set_output_port` sets the output port on the SUME board for packets whose Ethernet destination

address matches what was defined in the `forward` table; `compute_flow_id` computes the flow identifier of the packet to look up in the `retransmit` table; `cache_write`, `cache_read` and `cache_drop` modify `digest_data.tuser` to signal the cache queue to cache, retransmit and drop the packet at the head of the queue respectively.

3.2.3 The Extern Functions

P4 extern functions, or externs, are device-specific functions that are not described in the core P4 language—a kind of "black boxes" for P4 programs [35]. On NetFPGA, extern functions are implemented in HDL (Verilog). In other environments, they can be defined in any language, e.g. C++. The P4 program just sees the inputs and outputs, as parameters and results. There are two types of extern functions: stateless (reinitialised for each packet) and stateful (keeping states between packets). The stateful atomic externs are inspired by the Domino atoms [36]. P4→NetFPGA provides a set of commonly-used extern functions, shown in Table B.1.

An extern function can be declared using the syntax

```
extern void <name>_<extern_type>(in T1 data1,in T2 data2,...,out D result);
```

The following code sequence shows an example of declaring a simple longitudinal redundancy check hash extern:

```
@Xilinx_MaxLatency(1)
@Xilinx_ControlWidth(0)
extern void hash_lrc(in T in_data, out D result);
```

where `@Xilinx_MaxLatency` and `@Xilinx_ControlWidth` are Xilinx P4-SDNet's additional annotations to allow P4 programmer to specify the number of clock cycles to complete the extern operation, and the width of the address space allocated to this register respectively. The control width should always be equal to the width of the index field so that the control plane can access all register entries.

Using the retransmit logic in Figure 3.2 as the reference, I defined the following externs based on the externs template provided by P4→NetFPGA:

- `hash_lrc`: similar to P4→NetFPGA's LRC hash function, this splits the input data into multiple words and *XOR* each word together to obtain the result. This is used to hash the flow identifier of the packet. The hash result is used to index into the registers which store the counters for that particular flow.
- `seq_no_reg_praw`: stores the latest sequence number that the switch has seen from the sender. This is updated when a new packet is received.
- `latest_ack_no_reg_praw`: stores the latest acknowledgement number from the

receiver. This is updated when a new ACK packet is received.

- `pkts_cached_cnt_reg_raw`: stores the number of packets that the cache queue is buffering. This is updated when a new packet is buffered, or when some packets are read or dropped from the cache queue. When the latter happens, the value in the register is decremented either by 1 (read) or a positive value (dropped). The number of cached packets to drop is calculated by `(pkt.tcp.ackNo-latest_result) >> PKT_SIZE`. This limits the design to only be able to handle a single packet size that is configured in the P4 code. **For the purpose of easier debugging and testing process, I defined the packet size to be 64B**.
- `ack_cnt_reg_praw`: stores the number of duplicate acknowledgements. This is updated when the switch received a duplicate acknowledgement, that is when `pkt.tcp.ackNo == latest_ack_no`. When the value of the counter reaches 2, the third duplicate acknowledgement will signal the cache queue, via `digest_data.tuser`, to retransmit the packet with the sequence number in the DUP ACK packet. This packet will also be sent back to the sender with its ACK flag set to 0. This is because we do not want the sender to retransmit the packet, so an ACK flag of 0 keeps the counter of the sender at 2.
- `retransmit_cnt_reg_ifElseRaw`: stores the number of times of the DUP ACK packet has been retransmitted. Once a retransmission occurs, this counter will be set to 1. Subsequent DUP ACK packets will be sent back to the sender since the switch would now assume that the reason for packet loss is not due to a momentary failure, but rather to true congestion or inability to sustain the data rate.

The P4→NetFPGA `raw`, `praw` and `ifElseRAW` externs, by default, return the values of the register after modification. However, I often want to read the value of the register, *then* modify it. Thus, I changed their implementation so that they return the values before modification by having a temporary variable that stores the current value of the register. I also modified them to include the `SUBTRACT` operation.

Another important feature of P4→NetFPGA externs is that to guarantee consistency between successive packets, stateful operations cannot be pipelined; each performs an atomic read-modify-write operation per packet in hardware. In other words, each stateful extern can only be accessed *one* time in the P4 code. Multiple calls to the extern function will generate multiple instances of the atom, thus giving unexpected results. To illustrate this difficulty, let us look at an example of using a RAW extern (Read, add to, or overwrite state—Table B.1) called `curr_seq_no_raw` to store the latest sequence number that the switch has seen. To determine whether an incoming packet is a new packet, one first needs to perform a `READ` operation from the register to get the latest sequence number. If it is indeed a new packet, one then needs to perform a `WRITE` operation to update the value of the latest sequence number stored in the register. We can see, from the pseudocode in Figure 3.10, that we have made two calls

to the extern `curr_seq_no_raw`, which is not feasible with P4→NetFPGA externs.

```
def check_new_packet(pkt):
  curr_seq_no = curr_seq_no_raw(READ, 0) // READ from the register the
                                         // current latest sequence number
  if (pkt.seqNo > curr_seq_no):
    curr_seq_no_raw(WRITE, pkt.tcp.seqNo) // WRITE to the register the new
                                    // latest sequence number
                                    // Access the extern the SECOND time!!!
    return True
  else:
    return False
```

Figure 3.10: Pseudocode for checking whether an incoming packet is a new packet.

This complicates my P4 program significantly since we would naturally require two operations to perform most of the functions in the retransmit logic. Furthermore, some externs need to be called both when the packet is an ACK packet and when it is not, thus inevitably require at least two calls. To circumvent this, I used "pseudo-parameters", the PRAW extern (Either perform RAW or do not perform RAW based on predicate—Table B.1) and the ifElseRAW extern (Two RAWs, one each for when a predicate is true or false—Table B.1). The actual PRAW extern has eight parameters. For simplicity, let us define a simplified version of the PRAW extern, which has six parameters:

- `newVal`: the new value to write into the register.
- `operation`: the operation to perform (read, add to or write).
- `compVal`: the value to compare to the current value of the register.
- `comparator`: the operation used to compare `compVal` to the current value of the register. `compVal` and `comparator` together define the predicate that decides whether or not the register value will be modified. If the predicate is true then the register may be modified, otherwise it will not be. The predicate is defined as: <`compVal`> <`comparator`> <`register_value`>.
- `latest_value`: the current value of the register.
- `result`: indicates whether or not the predicate was `True` or `False`.

Continue from the previous example, our extern to access the register that stores the value of the latest sequence number that the switch has seen has to be called three times: when the packet is an ACK packet, we want to read the latest sequence number to see if the ACK number is the next sequence number; when the packet is not an ACK, we want to read the latest sequence number to determine if the packet is a new packet, and write back the updated latest sequence number if necessary. Now, using the pseudo-parameters, we can perform the three operations using only one call to the `curr_seq_no_praw` extern (Figure 3.11). If the packet is an ACK packet, `seq_no_compVal` is set to 0 and `comparator` is set to `GREATER_THAN`, so the predicate can never be true, since the value of `pkt.seqNo` is non-negative. The value of

`latest_seq_no_value` becomes the value of the register after the extern call. Thus, we essentially perform a `READ` operation. On the other hand, if the packet is not an ACK packet, `seq_no_compVal` is set to `pkt.seqNo`, and we now `WRITE` to the register if it is greater than the current latest sequence number. Otherwise, `latest_seq_no_value` contains the value of the register after the extern call, as above. The same technique is used for the ifElseRAW extern.

```
def access_sequence_number(pkt):
  result = False
  latest_seq_no_value = 0

  if pkt is an ACK pkt:
    operation = WRITE
    seq_no_compVal = 0
    comparator = GREATER_THAN
  else:
    operation = WRITE
    seq_no_compVal = pkt.seqNo
    comparator = GREATER_THAN

  // Access the extern ONE time -- OK!
  curr_seq_no_praw(pkt.tcp.seqNo, operation, seq_no_compVal,
              comparison, latest_seq_no_value, result);

  return (latest_seq_no_value, result)
```

Figure 3.11: Pseudocode for using "pseudo-parameters" to perform three operations using only one call to the extern.

With the actions and externs replacing the hash tables, our retransmit logic will now look like Figure 3.12.

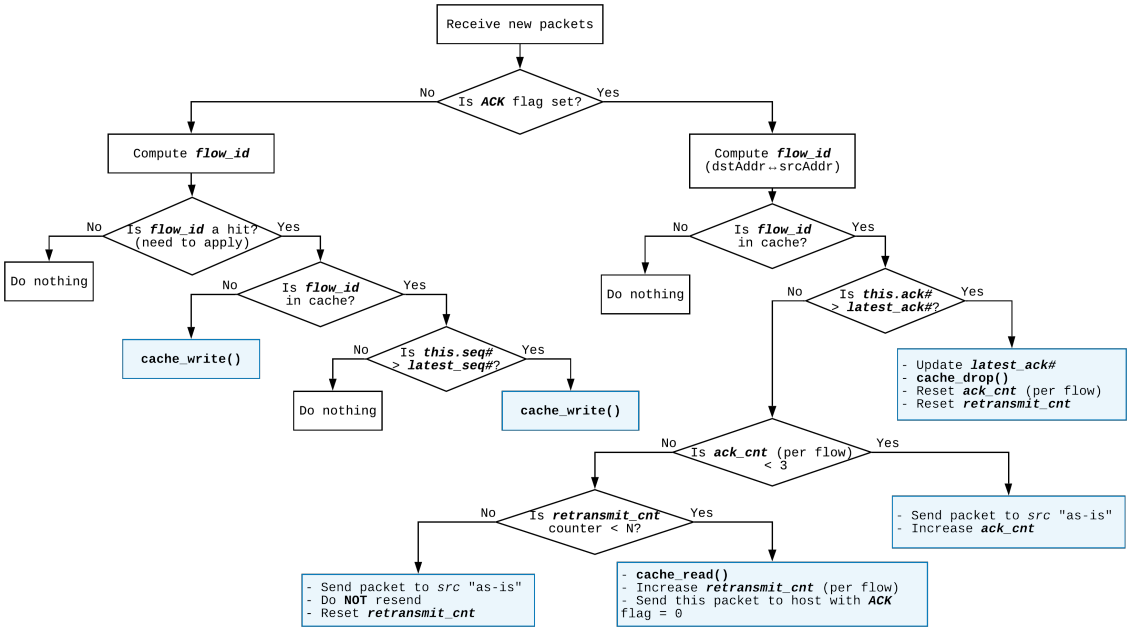


Figure 3.12: Flowchart of the packet retransmit logic using actions and externs. The steps involved are highlighted in blue box.

3.2.4 The Deparser

The inverse of parsing is deparsing, or packet assembly, where the outgoing packet is constructed by reassembling the packet headers as computed by the pipeline onto an outgoing packet byte stream. P4 does not provide a separate language for packet deparsing; deparsing is done in a `control` block that has at least one parameter of type `packet_out` because it only involves sequential logic as used for actions [18]. The advantage of this approach is that it makes deparsing explicit, but decouples it from parsing.

A header is added to the packet using the `packet_out` object's `emit` method. The following code block, which implements the deparser of the switch, first writes an Ethernet header, followed by an IPv4 header, and then a TCP header into a `packet_out`. Since emitting a header appends the header to the `packet_out` only if the header is valid, P4 first checks the validity of the headers before serialising them.

```
// Deparser Implementation
@Xilinx_MaxPacketRegion(8192)
control TopDeparser(packet_out b,
                    in Parsed_packet p,
                    in user_metadata_t user_metadata,
                    inout digest_data_t digest_data,
                    inout sume_metadata_t sume_metadata) {
  apply {
    b.emit(p.ethernet);
    b.emit(p.ip);
    b.emit(p.tcp);
  }
}
```

In summary, the switch will perform the following tasks in the SimpleSumeSwitch module:

1. Receive and parse packet from the sender. (Parser)
2. Look up the Ethernet destination address to determine the output port. Broadcast to all but the source port on a miss. (Match-action pipeline)
3. Compute the flow identifier of the packet. (Match-action pipeline)
4. Look up the flow identifier of the packet to determine if it should be monitored. Send normally on a miss. (Match-action pipeline)
5. Set `digest_data.tuser` and/or set the `ACK` flag in the TCP header appropriately. (Match-action pipeline)
6. Construct the final packet by appending the headers back and send it to the receiver. (Deparser)

3.3 Verilog Implementation

3.3.1 The Cache Queue

The cache queue has the basic functionalities similar to those of the output queue of the NetFPGA reference switch design (Figure 2.3): buffer packets from the SimpleSumeSwitch module while they wait to be sent to the output ports. However, since the role of the cache queue is to buffer packets to *retransmit*, we want to be able to signal the cache queue when to buffer a packet, when to drop a packet, when to read a packet and how many packets to drop. Recall from §2.7 that the standard metadata buses can be used to convey sideband data alongside each packet and allow P4-programmable elements to interact with non-programmable elements within the architecture. Hence, we can use the `digest_data` bus to convey our signal to the cache queue. Section §2.7 also says that the P4 programmer can define the format of the `digest_data` bus. The only constraint is that it *must* be defined to be 128-bit wide. Thus, to implement the signalling function to the cache queue, it is configured as follows:

```
struct digest_data_t {
  bit<72>  unused;
  bit<104> flow_id;
  bit<80> tuser;
}
```

The wider bus is used within the SimpleSumeSwitch module to carry sideband data between the parser, the match-action pipeline and the deparser. Here, we also use another 104 bits to store the flow identifier of the packet. The `digest_data` bus is then trimmed to the first 80 bits, and together with the `sume_metadata` bus form the 128-bit `tuser` bus. The `tuser` bus is one of the AXI-4 streams for inter-module communication (Figure B.2). It carries metadata between consecutive modules in the reference design. We use it to convey our signal from the SimpleSumeSwitch (P4-programmable module) to the cache queue module (non-programmable). Other AXI-4 streams are introduced in Table B.2. The `tready` and `tvalid` buses are also important in our cache queue design: `tready` signals to the preceding module that the current module is ready to receive the data, and `tvalid` indicates the validity of the data stream. The format of the `tuser` signal and the `digest_data` fields are shown in Table 3.3 and 3.4.

The output queue reads a packet at the start of the queue whenever it is not empty. With the `tuser` bus, we can now signal the cache queue to read or drop a packet by allowing the cache queue to read the packet only when the queue is not empty *and* there is a `cache_read` or `cache_drop` signal, then set the validity of that packet to 1 if the signal is a read signal, and 0 otherwise. An example for the first port `nf0` is given by the following pseudocode:

Table 3.3: Format of the `tuser` signal.

Bits	Name	Comments
[15:0]	`pkt_len`	Unsigned `int`
[23:16]	`src_port`	One-hot encoded: {DMA, NF3, DMA, NF2, DMA, NF1, DMA, NF0}
[31:24]	`dst_port`	One-hot encoded: {DMA, NF3, DMA, NF2, DMA, NF1, DMA, NF0}
[39:32]	`drop`	Only bit 32 is used
[47:40]	`send_dig_to_cpu`	Only bit 40 is used
[127:48]	`digest_data`	The first 80 bit of the `digest_data` bus from the SimpleSumeSwitch module

Table 3.4: Format of the `digest_data` field.

Bits	Name	Comments
[55:48]	`cache_write`	Encoded: {0, 0, 0, DMA, NF3, NF2, NF1, NF0}
[63:56]	`cache_read`	Encoded: {0, 0, 0, DMA, NF3, NF2, NF1, NF0}
[71:64]	`cache_drop`	Encoded: {0, 0, 0, DMA, NF3, NF2, NF1, NF0}
[79:72]	`cache_count`	Number of packets to read or drop
[127:80]	unused	-

```
assign nf0_tvalid = ~empty[0] & cache_read[0];
assign read_enable[0] = nf0_tready & ~empty[0]
                                   & (cache_read[0] | cache_drop[0]);
```

3.3.2 The Output Arbiter

Unlike standard NetFPGA designs (Figure 2.3), where each output port is fed by one output queue, here each output port may receive packets from two different queues: the output queue and the cache queue (Figure 3.3). Hence, the two imput streams need to be merged into one output stream before feeding to the output ports. This is because all input interfaces share the same bandwidth (and therefore width) as the output stream to ensure that maximum throughput can be achieved. The arbitration between the two queues, on packet level, is done by an arbiter. For this purpose, we added a new HDL module—inspired by NetFPGA's input arbiter—called the output arbiter before each of the five output ports.

Similar to the input arbiter, the output arbiter adopts a round-robin arbitration mechanism, alternating between the two queues. There is no priority between the five RX queues of the input arbiter, since their roles are equivalent. To keep the implementation simple, the output arbiter also imposes no priority between the two queues. Implementing the cache queue with a higher priority does not significantly improve the latency since it will mostly drop its packet. With round-robin, whenever it wants to send the packet, it waits for at most one packet from the output queue. The

output arbiter is also work-conserving, always trying to keep the output ports busy if there are packets in either the output queue or the cache queue. This ensures the packets get delivered as quickly as possible.

The difference between the output arbiter and the input arbiter is the number of input streams: the input arbiter has five input streams from five RX queues while the output arbiter has only two input streams from the output queue and the cache queue. Thus, I duplicated the implementation of the input arbiter in the NetFPGA reference switch design, removed the extra three slave stream ports and changed the appropriate parameters and wires to match the number of input streams.

3.3.3 The NetFPGA Datapath

The NetFPGA reference switch pipeline comprises different components (Figure 2.3), each of which is an HDL module. To implement the entire pipeline in hardware, we need to specify the datapath that connects the modules together in a separate Verilog file. The `nf_datapath.v` Verilog file (Appendix A) describes the hardware structure of the reference switch design. It specifies the logical circuit of the pipeline that allows for the automated analysis and simulation.

The NetFPGA reference switch pipeline is modified by adding the cache queue and the output arbiter HDL modules to the datapath and wiring the inputs and outputs of each module accordingly (Figure 3.3). The wiring process is tedious, due to the large number of inputs and outputs of each module.

3.3.4 IP Cores Generation

IP cores are Xilinx stand-alone modules that are configurable and reusable. They consist of the HDL module (Verilog/VHDL) and the tcl scripts to integrate them into the Vivado toolchain. After implementing the HDL modules, I wrote the tcl files for the cache queue and the output arbiter, declaring the hardware resources and libraries used and setting various properties and parameters of the modules. I then modified two other tcl scripts, `simple_sume_switch.tcl` and `simple_sume_switch_sim.tcl`, which are NetFPGA's example project-level scripts to include the tcl scripts of the cache queue and the output arbiter. These scripts then build the project by defining the connectivity between the modules, generating the cores, performing static timing analysis and compiling the design into bitstream.

Summary

I have implemented the core logic of the switch in P4 code (§3.2) and the Verilog components of the switch. With that, I have successfully implemented the architecture described in §3.1. Moreover, with the P4 implementation, I have met my first success criterion.

Chapter 4

Evaluation

The objective of this chapter is to review my implementation's success at meeting the original project success criteria. Hence, I start by discussing the project success criteria and how my project device meets them (§4.1). Then, I assess my implementation according to the following goals:

- **Correctness**: The correctness of the design is assessed using simulation-based verification tests to look for errors (§4.2).
- **Functionality**: The functionality of the device is assessed using multiple simulations and hardware-based functional tests (§4.3).
- **Performance**: The performance of the design is evaluated in various aspects, which are listed in §4.4. The analysis also compares the design's performance with that of a standard switch, where appropriate.

Finally, I demonstrate the device's interoperability (§4.4.2) and discuss the limitations of the current design (§4.4.3).

4.1 Overall Results

The success criteria of the project, as described in the original proposal (Appendix D) and refined in §2.3, have all been met. They are briefly summarised below, with pointers to the relevant discussion in this document:

- **Criterion 1**: *Having an implementation of the switch's functionalities in P4 code.*

The P4 code constituted the basis of the project and was completed in the early stage. It was revised many times, based on the code review feedback from my supervisor and the results of simulations and tests. The P4 implementation was described in detail in §3.2, while the hardware implementation, which provides the basis to meet Criteria 2 and 3, was discussed in §3.3.

- **Criterion 2**: *The implementation passes two different simulations.*

Simulation and testing is the primary subject of this chapter. The design was simulated through various test cases and passed two different types of simulation: block-level and chip-level simulation. The details are discussed in §4.2, including the test data generation process and the test cases chosen.

• **Criterion 3**: *The implementation works correctly in hardware (functional system test).*

After passing the simulations, the design was compiled and tested on the NetFPGA SUME board. The results are presented in §4.3.

• **Criterion 4**: *Able to demonstrate its interoperability with a software-based application.*

Its interoperability was demonstrated in §4.4.2 using two simple applications: `ping` and `iperf3`.

• **Criterion 5**: *Provide a performance evaluation of the design.*

The performance of the design is measured using three metrics: latency, throughput and resources utilisation. It is then analysed in comparison to the performance of a standard switch written in Verilog [37, 38] and evaluated. The results are presented in §4.4.

• **Extension 1**: *Sending a notification to the source if multiple retransmissions fail.*

With the current system architecture, the switch will retransmit *once* if it receives three DUP ACKs. If the retransmission fails, subsequent DUP ACK packets will be sent back to the sender normally. This was implemented using the `retransmit_cnt_reg_ifElseRaw` extern, which was discussed in §3.2.3.

• **Extension 2**: *Provide an evaluation in comparison to a standard switch performance.*

A standard switch design [37, 38] was also tested for performance using the same metrics. The results are used to compare with the project design performance in §4.4.1 and §4.4.3, where the difference in the performance is also discussed.

4.2 Simulation Environment

It is generally easier to debug program behaviour in simulation than in hardware, and the purpose of the simulation environment is exactly that. For this purpose, the P4→NetFPGA workflow provides a simulation environment with automated test infrastructure. There are two types of simulation: block-level simulation (SDNet

simulation) and chip-level simulation (SUME simulation). The SDNet simulation covers the correctness of the P4 implementation on packet-level. It checks that the output port and the correct signal to the cache queue were set correctly in the packet's `sume_metadata` and `digest_data` respectively. The SUME simulation, on the other hand, tests scenarios that cannot be covered by the SDNet simulation. For example, when the third DUP ACK packet is received, the P4 code modifies `digest_data.tuser` to signal the cache queue to retransmit the packet in port `nf1`. In SDNet simulation, we only expect to see one packet coming out of port `nf0` with its `digest_data.tuser` modified. However, in SUME simulation, we expect to observe the retransmitted packet as well. Thus, even if the inputs are the same as the SDNet simulation, the expected outputs and functionalities are different in the SUME simulation.

To run a simulation, we provide the corresponding test bench with the appropriate scripts. We will modify the following example files, which are provided by the P4→NetFPGA workflow:

For **SDNet simulation**:

- `commands.txt`: used to generate the control plane functionality. As such, it contains the set of commands to add entries to the match-action tables that we have defined in our P4 program (see Table 3.1 and 3.2). These entries will be used in the SUME simulation and by the SUME board.
- `gen_testdata.py` uses `scapy` library to generate test packets (& metadata), along with the corresponding expected output packets and metadata. P4→NetFPGA compares the output file with the expected output to give the result. This script also generates `.pcap` files from the test packets that will be used in the SUME simulation.

For **SUME simulation**:

- `nf_test.py` runs the SUME simulation by executing `run.py`.
- `run.py` reads the packet traces (`.pcap` files) generated by the `gen_testdata.py` script and/or creates new packets and applies them to the SUME interfaces. It also defines a test sequence, including configuration and timing stimulus.

4.2.1 Test Data Generation

The `gen_testdata.py` template provided by the P4→NetFPGA workflow has two functions—`applyPkt()` and `expPkt()`—which allow us to specify input packets and expected output packets respectively. I wrote the `digest_data.py` module and used Python `scapy` module to generate the metadata and test packets. Figure 4.1 shows how to create and send a test packet, and specify the expected packet. We create the packet

headers using `scapy`'s `Ether`, `IP` and `TCP` classes, stacking the layer with the `/` operator and pad it to 64 bytes with the `pad_pkt()` method. Then, `applyPkt()` will "send" the packet to the SimpleSumeSwitch module. Now, we need to specify how we would expect the output packet. We create two variables, `flow_id` and `actions`, to represent the values of the flow identifier and the `digest_data.tuser` field of the output packet. Finally, we use `expPkt()` to put the packet into a list of "expected" packets.

```
from scapy.all import *
import digest_data

MAC_src = "08:11:11:11:11:08" # Source MAC addr.
MAC_dst = "08:22:22:22:22:08" # Destination MAC addr.
sport = 55 # Source L4 port
dport = 75 # Destination L4 port
IP_src = "10.0.0.1" # Source IP addr.
IP_dst = "10.0.0.2" # Dest IP addr.

pkt = (
  Ether(src=MAC_src, dst=MAC_dst)
  / IP(src=IP_src, dst=IP_dst)
  / TCP(sport=sport, dport=dport, flags="S", seq=1)
)                               # create the packet using scapy Ether, IP
                                # and TCP classes.
pkt = pad_pkt(pkt, 64)          # pad the packet to 64 bytes
applyPkt(pkt, "nf0", 0)         # send from port 0

# compute the flow number
flow_id = digest_data.compute_flow_number(IP1_src, IP1_dst,
                6, sport, dport)

# write to port 1 of cache_queue
actions = digest_data.compute_tuser(0, 0, 0,
                tuser_map["nf1"])

# expect from port 1 of output_queue
expPkt(pkt, "nf1", drop=False, flow_id=flow_id, tuser=actions)
```

Figure 4.1: A Python code snippet to create test packets and specify the expected output.

To assess the correctness of the P4 implementation, I used the five following tests in sequence:

Test #1: *Basic packet forwarding of the SimpleSumeSwitch module.*

1. **Goal**: Test that the program correctly forwards a packet and modifies its metadata.
2. **Input**: Send 1 packet from port `nf0` to port `nf1`. This packet belongs to a flow of interest.
3. **Expected output**: Expect 1 packet coming out of port `nf1` with the `digest_data.flow_id` field computed and matches the flow identifier of the input packet, and `digest_data.tuser` set to signal the cache queue to buffer the packet in port `nf1`.

Test #2: *Program behaviour when standard ACK packets are received.*

1. **Goal**: Test that the program correctly signals the cache queue to drop buffered

packets that have been acknowledged.
2. **Input**: Send 3 packets, all of which belong to a flow of interest, from port `nf0` to port `nf1`. Send 1 ACK packet to port `nf0` acknowledging the receipt of the first packet. Send the 2^{nd} ACK packet to acknowledge the receipt of the remaining two packets.
3. **Expected output**: Expect 2 ACK packets coming out of port `nf0`. The first packet has `digest_data.tuser` set to drop 1 packet from port `nf1`. The second packet has `digest_data.tuser` set to drop the remaining 2 packets from port `nf1`.

Test #3: *Program behaviour when three DUP ACK packets are received.*

1. **Goal**: Test that the program correctly signals the cache queue to **retransmit** the required packet.
2. **Input**: Send 1 packet from port `nf0` to port `nf1`. Send 3 DUP ACK packets from port `nf1` to port `nf0` requiring for the first packet.
3. **Expected output**: Expect the third DUP ACK packet coming out of port `nf0` with its ACK flag set to 0, and `digest_data.tuser` set to signal the cache queue to resend the packet in port `nf1`.

Test #4: *Program behaviour when a fourth DUP ACK packet is received.*

1. **Goal**: Test that the program correctly sends the subsequent DUP ACK packets back to the sender normally (i.e. only retransmit once).
2. **Input**: Send 1 packets, all of which belong to a flow of interest, from port `nf0` to port `nf1`. Send 1 ACK packet acknowledging the first packet. Send the 2^{nd} ACK packet to acknowledge the remaining two packets.
3. **Expected output**: Expect the ACK packet coming out of port `nf0` to be sent back to the sender without any modification.

Test #5: *Program behaviour when there are multiple flows, only one of which is a flow of interest.*

1. **Goal**: Test that the program correctly monitors the interested flow (of a latency-sensitive application) and delivers packets of other flows normally.
2. **Input**: Create 3 flows and randomly interleave their packets. Send them from port `nf0` to `nf1`.
3. **Expected output**: Expect to see that the packets from our interested flow are modified accordingly—buffered, dropped or retransmitted—while the other packets are delivered normally.

After creating the packets and specifying the expected outputs, we proceed to simulate our design.

4.2.2 SDNet Simulation

First, we compile our P4 code with the SDNet compiler. After successful compilation, we run the SDNet simulation using Vivado Simulator (§2.7). The simulator will "send" the user-defined input packets and metadata to the SimpleSumeSwitch HDL module and compare the packets at the output of the P4 switch with the expected packets we specified in `gen_testdata.py`.

A total of 29 packets were used in the tests. Figure 4.2 shows all the packets are received and their metadata are modified as we expected. If the simulation output results in a metadata error (i.e. a mismatch between expected and actual tuples), we can debug using a script to parse the metadata into more readable form and check which fields do not match (Figure 4.3).

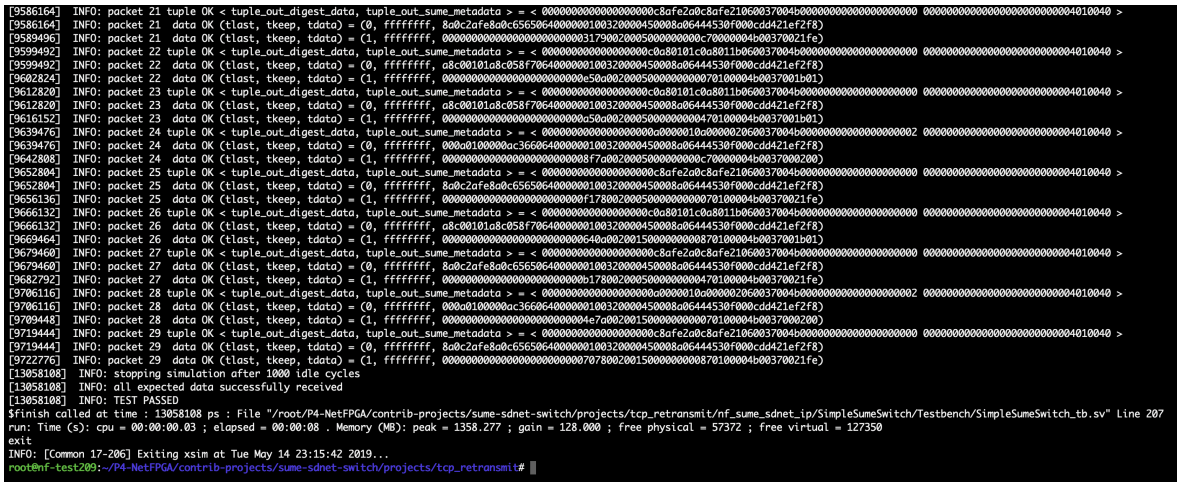


Figure 4.2: Test output of the SDNet simulation using Vivado Simulator showing success.

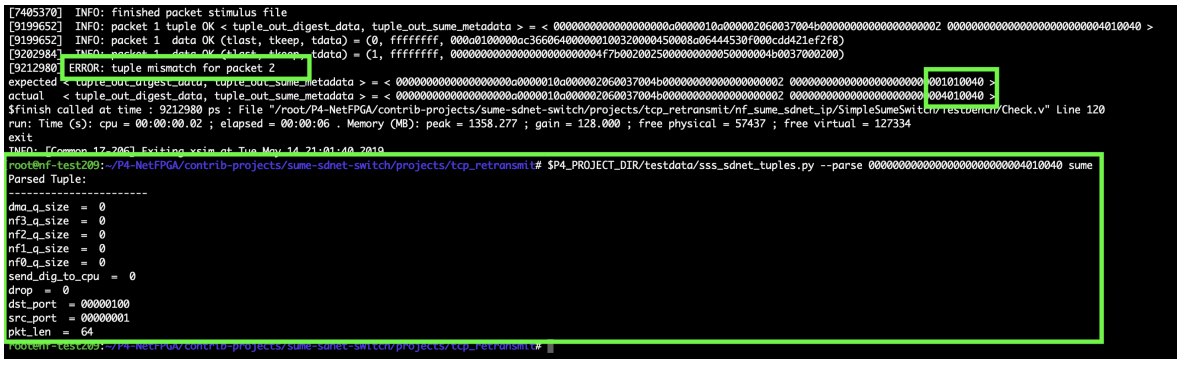


Figure 4.3: Test output of SDNet simulation showing error. Using a Python script to parse the metadata into more readable form, we can debug our program.

4.2.3 SUME simulation

The SUME simulation installs the SimpleSumeSwitch HDL module as a NetFPGA IP core and simulates the behaviour of the entire NetFPGA reference switch design.

This way, it can cover those scenarios that involve the cache queue, which the SDNet simulation cannot.

I used the same stimuli as the SDNet simulation to verify that the SimpleSumeSwitch module was successfully integrated into our modified reference switch pipeline and assess its correctness. The outcome for the tests are mostly the same. The only noticeable difference is in **Test #4**, where we received a third DUP ACK packet. We would expect the cache queue to retransmit the packet in port `nf1`, hence an extra packet. Figure 4.4 shows that we indeed receive the retransmitted packet, which makes the total number of packets 30. Figure 4.5 illustrates this further by showing the traces of packets leaving port `nf0` and `nf1` of the output queue and the cache queue. There is one packet that coming out of port `nf1` of the cache queue, which is the retransmitted packet.

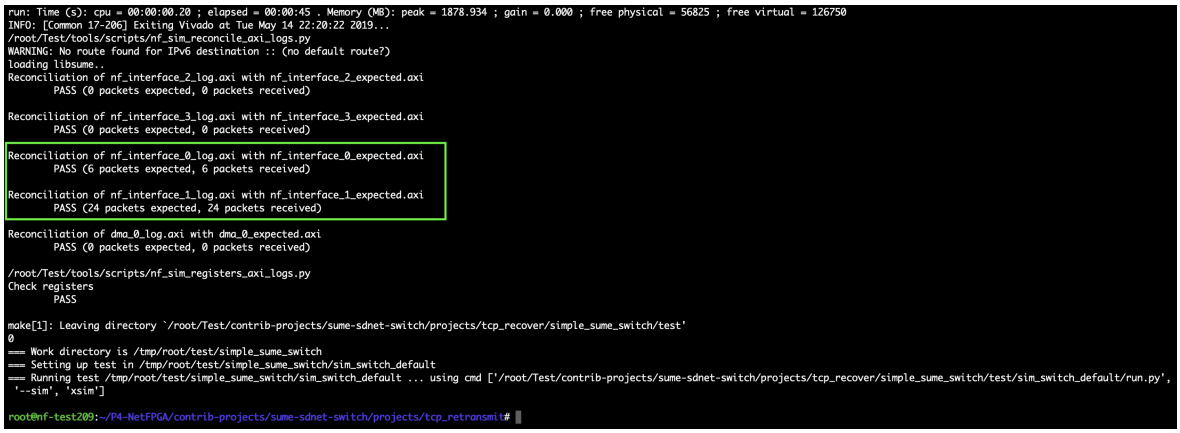


Figure 4.4: Test output of the SUME simulation using Vivado Simulator showing success. We now receive 30 packets instead of 29.

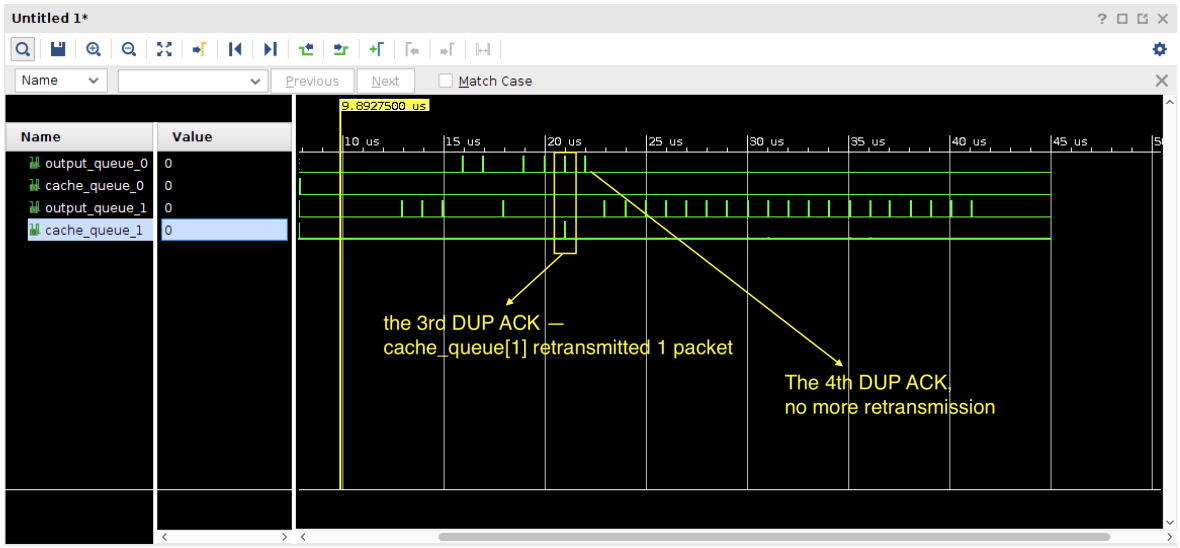


Figure 4.5: Snapshot of Vivado Design Suite GUI showing the traces of the packets coming out of port `nf0` and `nf1` of both the output queue and the cache queue. There is a retransmission on the third DUP ACK packet, but none on the fourth.

4.3 Hardware Test

After all simulations indicated that the P4 implementation is working correctly, I built the FPGA bitstream and tested the functionality of my design on the NetFPGA SUME board. The NetFPGA platform enables running the same tests as in simulations on hardware, with the exception of the limited number of NIC ports (2). Since the number of NIC ports does not affect my design's behaviour, I decided to use the same tests from the SUME simulation.

The setup for the hardware test requires a machine with the NetFPGA SUME board and a 10Gb NIC, connected as shown in Figure 4.6. The switch passed the hardware tests, as shown by the test output log in Figure 4.7, indicating that the switch is fully functional.

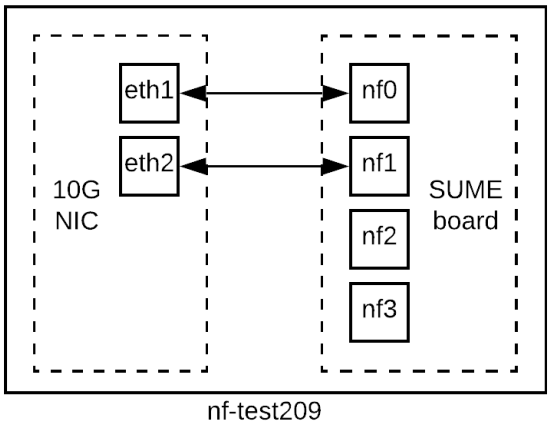


Figure 4.6: The setup for the hardware test.

```
Root directory is /root/P4-NetFPGA
   Running global setup...  PASS
   Running test both_switch_default... PASS
loading libsume..
Running test using the following physical connections:
nf3 initialized but not connected
nf2 initialized but not connected
nf1:eth2
nf0:eth1
------------------------------------------------------
About to start the test
Pass: device nf1 saw 24 expected packets
Pass: device nf0 saw 6 expected packets
Pass: device eth2 saw 24 expected packets
Pass: device eth1 saw 6 expected packets
------------------------------------------------------
   Running global teardown...  PASS
root@nf-test209:~/P4-NetFPGA#
```

Figure 4.7: Successful test output of the hardware test.

4.4 Performance Evaluation

4.4.1 Performance Analysis

I used three metrics to evaluate the performance of the switch: latency, throughput and resources used. The latency and throughput were measured *by my supervisor* on a dedicated setup in the Computer Laboratory data center, to which I have limited access. The setup, as shown in Figure 4.8, includes one machine with my design programmed onto the SUME board connecting to a second machine with a SUME board that has Open Source Network Tester [39, 40] (OSNT) installed on it. OSNT is a fully open-source traffic generator and capture system that is compatible with the NetFPGA-SUME platform. The same setup is also used for unified performance

evaluation of other projects, including all Part III P51 course[12] projects.

Latency

The latency of the switch was measured by sending 1000 packets of various sizes, ranging from 60B to 1514B, and measuring the round-trip time (RTT) of each packet. The values are then adjusted to compensate for the latency of the setup. The similar experiment was performed on a standard reference switch, and the two results are shown in Figure 4.9, which plots the average latency against packet size.

As we would expect from a P4 store and forward switch, the latency increases with increasing packet size, ranging from 3250ns to 5340ns. In comparison, my device takes 2500ns longer than the standard switch (i.e. without the cache queue) written in Verilog. This is reasonable since we would expect the retransmit logic to add some overhead to the packet processing time.

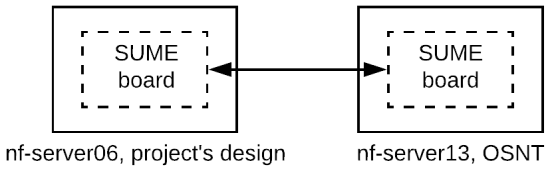


Figure 4.8: Setup to measure the latency and throughput of the switch.

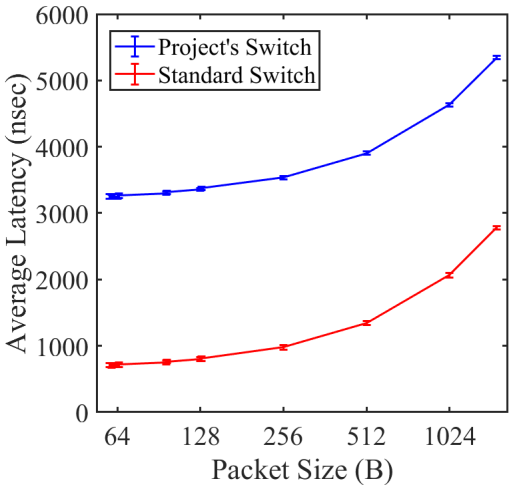


Figure 4.9: The average latency of the project's device in comparison with the standard switch. Both are measured by sending 1000 packets of various sizes.

Throughput

The throughput of the switch was measured on 1 port (Figure 4.10a) and 4 ports (Figure 4.10b) running at the full speed of **10Gbps/port** (**40Gbps per design**). In the first case, using 64-Byte packets, packet drops occurred when 100 million packets were sent, while that for 65-Byte packets is 1 billion. Similar results were achieved in the latter case, where packet losses occurred only when 1 billion packets were sent to each of the 4 ports.

[12] https://www.cl.cam.ac.uk/teaching/1819/P51/

As demonstrated in Figure 4.10, my device achieved 99.9999% throughput most of the time. No confidence interval was given because the error is negligible (3 packets out of 1 billion) and the test was performed once. Ideally, no packet loss is desirable. Nevertheless, the results are reasonable and expected, taking into account subtle clock differences between OSNT and the device under test.

Pkt. size	Pkts. sent	Pkts. dropped
64B	10 M.	0
	100 M.	15
	1 B.	3
65B	10 M.	0
	100 M.	0
	1 B.	3

(a) Throughput test on 1 port, using packet size of 64B and 65B.

Pkt. size	Pkts. sent	Pkts. dropped
64B	10 M.	0
	100 M.	0
	1 B.	1767

(b) Throughput test on 4 ports, using packet size of 64B.

Figure 4.10: Throughput of the switch testing with 1 port **(a)** and 4 ports **(b)** running at full speed of 10 Gbps/port. B: Bytes, M.: Million, B.: Billion.

Resources Used

When we compiled the NetFPGA bitstream, Vivado synthesised and implemented the design, and provided us with a summary of the timing analysis and resources usage. Figure 4.11 shows that our implementation *passed* the timing analysis—all the timing constraints we imposed are satisfied. Figure 4.12 shows the utilisation statistics of two of the resources—the memory and the LUTs—of both devices. We can see that our device used more resources than the reference switch: more than 30% LUTs and 60% memory. This is likely due to the larger number of computations and registers used by our P4 program.

```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------

    WNS(ns)   TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoi
nts
   -------   -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------  ----------------------  -----------------
---
      0.19     0.000                      0                81242        0.054        0.000                      0                81231       0.000        0.000                       0                39
572

All user specified timing constraints are met.
```

Figure 4.11: An extract from the Timing Analysis Report of Vivado showing our design met all the timing constraints.

4.4.2 Interoperability

To demonstrate the interoperability of the device, we connect it between the two 10Gb NIC of two machines, as shown in Figure 4.13. Then, we run some simple applications between the two machines and observe if the switch is able to deliver the packets. In this project, I demonstrated using `ping` and `iperf3`. In the first setup, we ping

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	140222	0	433200	32.37
LUT as Logic	109201	0	433200	25.21
LUT as Memory	31021	0	174200	17.81
LUT as Distributed RAM	6424	0		
LUT as Shift Register	24597	0		
Slice Registers	334654	0	866400	38.63
Register as Flip Flop	334650	0	866400	38.63
Register as Latch	4	0	866400	<0.01
F7 Muxes	1300	0	216600	0.60
F8 Muxes	130	0	108300	0.12

(a) Slice logic utilisation of project device.

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	977	0	1470	66.46
RAMB36/FIFO*	933	0	1470	63.47
FIFO36E1 only	8			
RAMB36E1 only	925			
RAMB18	88	0	2940	2.99
RAMB18E1 only	88			

(b) Memory utilisation of project device.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	64657	0	433200	14.93
LUT as Logic	59547	0	433200	13.75
LUT as Memory	5110	0	174200	2.93
LUT as Distributed RAM	2966	0		
LUT as Shift Register	2144	0		
Slice Registers	111950	0	866400	12.92
Register as Flip Flop	111949	0	866400	12.92
Register as Latch	0	0	866400	0.00
Register as AND/OR	1	0	866400	<0.01
F7 Muxes	714	0	216600	0.33
F8 Muxes	25	0	108300	0.02

(c) Slice logic utilisation of reference switch.

3. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	454.5	0	1470	30.92
RAMB36/FIFO*	439	6	1470	29.86
FIFO36E1 only	8			
RAMB36E1 only	431			
RAMB18	31	4	2940	1.05
RAMB18E1 only	31			

(d) Memory utilisation of reference switch.

Figure 4.12: Specialised resources usage extracted from the device utilisation statistics report of Vivado.

from `10.0.0.1` to `10.0.0.2` and capture the packets on `10.0.0.2` with `tcpdump` for visualisation. In the second setup, we run `iperf3` server on `10.0.0.2` and client on `10.0.0.1`. The outputs of both setups are shown in Figure 4.14 and 4.15 respectively, indicating the interoperability of the device. I should note that the numbers in Figure 4.15 may well be an underestimate of my device's capability. This could be due to `iperf3` trying to send packets of size larger than 64B, while my device only handles a single packet size of 64B.

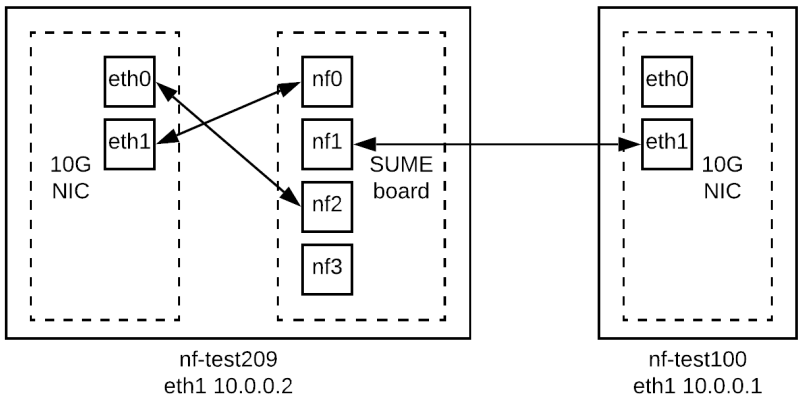


Figure 4.13: Setup to demonstrate the interoperability of the switch.

```
root@nf-test209:~# tcpdump -i eth1 -c 5 host 10.0.0.1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:24:20.834599 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 28710, seq 1, length 64
20:24:20.834616 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 28710, seq 1, length 64
20:24:20.834803 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 28710, seq 2, length 64
20:24:20.834807 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 28710, seq 2, length 64
20:24:20.834864 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 28710, seq 3, length 64
5 packets captured
30 packets received by filter
13 packets dropped by kernel
root@nf-test209:~#
```

Figure 4.14: Output of running `tcpdump` on `10.0.0.2` capturing packets to and from `10.0.0.1`.

```
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 10.0.0.1, port 35338
[  5] local 10.0.0.2 port 5201 connected to 10.0.0.1 port 35340
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00   sec  88.1 KBytes  88.1 KBytes/sec
[  5]   1.00-2.00   sec   148 KBytes   148 KBytes/sec
[  5]   2.00-3.00   sec   144 KBytes   144 KBytes/sec
[  5]   3.00-4.00   sec  62.4 KBytes  62.4 KBytes/sec
[  5]   4.00-5.00   sec  92.2 KBytes  92.2 KBytes/sec
[  5]   5.00-6.00   sec   103 KBytes   103 KBytes/sec
[  5]   6.00-7.00   sec   125 KBytes   125 KBytes/sec
[  5]   7.00-8.00   sec   115 KBytes   115 KBytes/sec
[  5]   8.00-9.00   sec   111 KBytes   111 KBytes/sec
[  5]   9.00-10.00  sec   249 KBytes   249 KBytes/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  5]   0.00-10.00  sec  2.02 MBytes  205 KBytes/sec    0             sender
[  5]   0.00-10.00  sec  1.33 MBytes  137 KBytes/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
root@nf-test209:~#
```

Figure 4.15: Output from the `iperf3` server machine.

4.4.3 Limitations

The design has been assessed for correctness (via simulations), functionality (via functional system test), performance and interoperabilty. While the results have been positive, I have not been able to demonstrate the performance improvement of the switch under an interoperability scenario. In other words, I have also attempted to measure the time in performing real tasks such as file transferring (using `scp`) with and without my design, but there is no signifcant difference in the performance.

Chapter 5

Conclusion

5.1 Accomplishments

This dissertation has described the design, implementation and evaluation of using a programmable switch to fast retransmit TCP packet losses that are not due to congestion. Overall, the project achieved its original aims as the switch functionalities passed the simulations for correctness and behaved correctly in hardware. Its performance has also been analysed and evaluated, where it performed reasonably close to a standard switch in various aspects.

Personally, I have enjoyed this fascinating opportunity to enhance my knowledge in the field of computer networking, especially high-performance networks. I undertook this project to further comprehend and appreciate the intricacy of TCP congestion control mechanism, the potential of data plane programmability and how they can be used together to improve cross-datacentre performance.

Finally, this project also contributed to my personal development by improving my software engineering and technical writing skills.

5.2 Lessons Learnt

The process of transforming an initial idea into a working programmable switch was extremely enjoyable. However, a considerable amount of time was spent on learning the languages, the development environments and their limitations, and designing the architecture. With the benefit of hindsight, I would have familiarised myself with the languages and tools, and implemented the architecture prior to starting the project, then used those as the starting point. This would have given me more time to focus on testing and evaluating the design, and explore useful extensions, so that the programmable switch can improve the performance with real traffic.

5.3 Future Work

The project is complete in the sense that it satisfies the initial project requirements. Nevertheless, there are many promising avenues for further improvement that were not explored due to time constraints. These include:

- **Supporting multiple packet sizes**. The current design only supports a single packet size, defined at compile time. It would definitely be more useful if the design could support multiple packet sizes dynamically, without first specifying them.
- **Supporting multiple flows**. It would also be useful to have one programmable switch to serve different applications between the same sender and receiver. With only one cache queue available, the current design only supports one single flow. Adding more cache queues would require a more complicated mechanism to signal a particular cache queue.
- **Dynamic configuration**. The configurations of the flow and the packet size are currently pre-defined and embedded within the P4 code. A more flexible design could allow the user adaptively installing and removing flows to monitor, as well as to configure the flow.

Bibliography

[1] N. Zilberman, M. Grosvenor, D. A. Popescu, N. Manihatty-Bojan, G. Antichi, M. Wójcik, and A. W. Moore, "Where has my time gone?" in *International Conference on Passive and Active Network Measurement*. Springer, 2017, pp. 201–214.

[2] Sapio, Amedeo, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 150–156.

[3] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, N. Zilberman, and R. Soulé, "P4xos: Consensus as a Network Service," ser. Tech Report, May 2018. [Online]. Available: http://web.inf.usi.ch/file/pub/105/p4xos.pdf

[4] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," vol. 46, no. 2, pp. 18–24, May 2016.

[5] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '15. Renton, WA, USA: ACM, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-jin.pdf

[6] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132764

[7] Noa Zilberman, "In-Network Computing," https://www.sigarch.org/in-network-computing-draft/.

[8] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC Editor, RFC 2001, January 1997. [Online]. Available: https://tools.ietf.org/html/rfc2001

[9] E. Blanton, V. Paxson, and M. Allman, "TCP Congestion Control," RFC Editor, RFC 5681, September 2009. [Online]. Available: https://tools.ietf.org/html/rfc5681

[10] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC Editor, RFC 3168, Septermber 2001. [Online]. Available: https://tools.ietf.org/html/rfc3168

[11] Bensley, S. and Thaler, D. and Balasubramanian, P. and Eggert, L. and Judd, G., "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers," RFC Editor, RFC 8257, October 2017. [Online]. Available: https://tools.ietf.org/html/rfc8257

[12] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 523–536. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787484

[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 503–514. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626316

[14] M. A. Attar, J. Crowcroft, L. Eggert, and K. Wehrle, "Network Latency Control in Data Centres (Dagstuhl Seminar 16281)," *Dagstuhl Reports*, vol. 6, no. 7, pp. 15–30, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6760

[15] Simon Jouet, "Programmable Data Plane — The Next Step in SDN?" https://www.geant.org/Services/Connectivity_and_network/GTS/Documents/Simon%20Jouet%20-%20Programmable%20Dataplane.pdf.

[16] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[18] "P4 language spec version 1.0.0-rc2." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html

[19] "The P4 Language Consortium," https://p4.org/.

[20] "Protocol Oblivious Forwarding (POF) Website." [Online]. Available: http://www.poforwarding.org/

[21] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 127–132.

[22] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "Dc. p4: Programming the forwarding plane of a data-center switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2015, p. 2.

[23] C. Kim, A. Sivaraman, N. P. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," 2015.

[24] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[25] Philippe Biondi, "Scapy," 2019. [Online]. Available: https://scapy.net/

[26] Alan Blackwell, "Software and Interface Design," 2015, University of Cambridge Part IA CST course notes.

[27] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4→NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293924

[28] "P4→NetFPGA Workflow Overview," https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview.

[29] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," in *IEEE Micro34-1*, 2014, pp. 8–18.

[30] N. Sultana *et al.*, "Emu: Rapid prototyping of networking services," in *USENIX Annual Technical Conference*, 2017.

[31] Ross Anderson, "Software Engineering," 2017, University of Cambridge Part IA CST course notes.

[32] "NetFPGA/P4-NetFPGA-public Wiki Page," https://github.com/NetFPGA/P4-NetFPGA-public/wiki.

[33] "NetFPGA SUME Beta List," https://lists.cam.ac.uk/mailman/listinfo/cl-netfpga-sume-beta.

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[35] "CS344–Lecture 2: P4 Language Overview," https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview.

[36] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 15–28.

[37] "NetFPGA SUME Reference Learning Switch." [Online]. Available: https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-Learning-Switch

[38] "Reference Switch bitfile." [Online]. Available: http://www.cl.cam.ac.uk/research/srg/netos/projects/netfpga/bitfiles/NetFPGA-SUME-live/1.9.0/reference_switch/reference_switch.bit

[39] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, G. A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: Open Source Network Tester," *IEEE Network*, vol. 28, pp. 6–12, 2014.

[40] "OSNT SUME Home," https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-SUME-Home.

[41] Jacob Leverich, "Mutilate: High-performance memcached load generator," https://github.com/leverich/mutilate.

[42] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "Network Hardware-Accelerated Consensus," USI, Research Report 2016-03, May 2016. [Online]. Available: http://www.inf.usi.ch/faculty/soule/usi-tr-2016-03.pdf

[43] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," vol. 34, no. 1, pp. 8–18, Jan. 2014.

Appendix A

Repository Overview

The project repository comprises four main folders:

```
P4-NetFPGA
├── contrib-projects
├── docs
├── lib
└── tools
```

- `contrib-projects`: contains the reference switch pipeline, the core logic of the switch in P4 and the testing scripts.
- `docs`: contains design documentations and user-guides.
- `lib`: contains custom and reference IP Cores and software libraries.
- `tools`: contains scripts for automations: running simulations, etc.

The `contrib-projects`, `lib` and `tools` folders contain the P4 (*) and Verilog (**) implementation of the design:

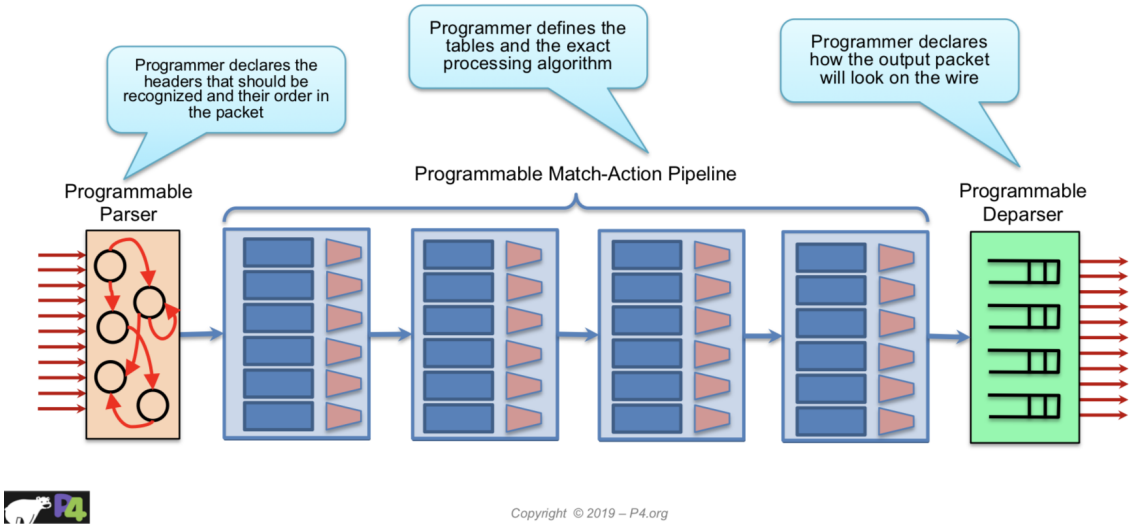
```
contrib-projects
├── sume_sdnet_switch/projects/tcp_retransmit
│   ├── simple_sume_switch/hw
│   │   ├── hdl/nf_datapath.v**
│   │   └── tcl
│   │       ├── simple_sume_switch.tcl
│   │       └── simple_sume_switch_sim.tcl
│   └── test/sim_switch_default
│       └── run.py**
├── src
│   ├── tcp_retransmit.p4*
│   └── commands.txt*
├── testdata
│   ├── gen_testdata.py*
│   ├── digest_data.py*
│   └── sss_sdnet_tuples.py*
└── templates
```

```
    └── externs/<externs_type>/hdl/<externs_type>_template.v**

lib
└── hw
    ├── contrib/cores
    │   └── sss_cache_queues_v1_0_0**
    └── std/cores
        └── output_arbiter_v1_0_0**

tools
└── scripts/NFTest/nf_test.py**
```

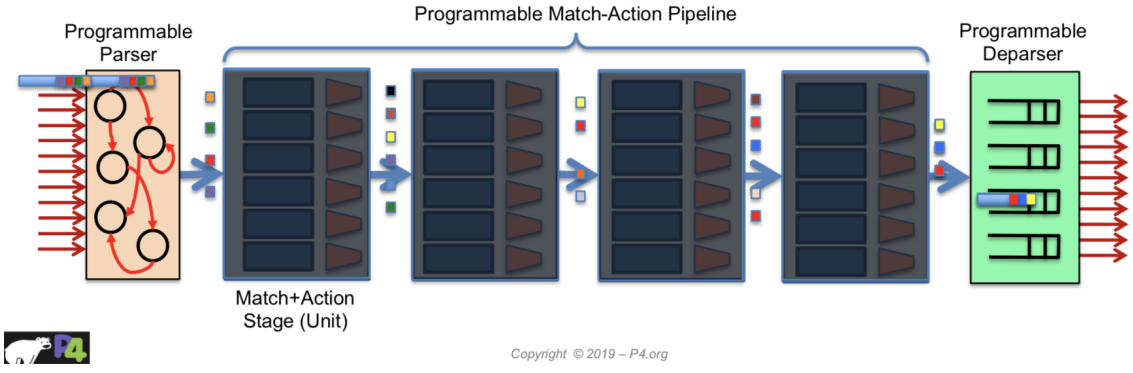
Appendix B

Figures and Tables

Appendix B contains Figures and Tables that are not specific to the project, but were referred to in the dissertation.



(a) The single programmable pipeline forwarding architecture of PISA.



(b) The life cycle of a packet going through the PISA pipeline.

Figure B.1: PISA—Protocol-Independent Switch Architecture. Source: P4.org – Copyright © 2019.

Table B.1: The P4→NetFPGA extern functions library.

Stateful Atomic Extern Functions	
Type	**Description**
RW	Read or write state
RAW	Read, add to, or overwrite state
PRAW	Either perform RAW or do not perform RAW based on predicate
ifElseRAW	Two RAWs, one each for when a predicate is true or false
Sub	IfElseRAW with stateful subtraction capability
Stateless Extern Functions	
Type	**Description**
IP Checksum	Given an IP header, compute the IP checksum
LRC	Longitudinal redundancy check, simple hash function
timestamp	Generate timestamp (measured in clock cycles, granularity of 5ns)

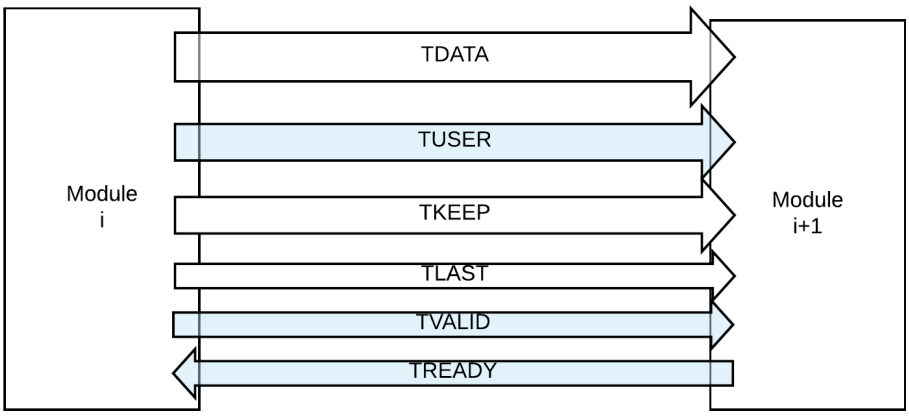


Figure B.2: Inter-module communication is done via AXI-4 streams (Packets are moved as stream).

Table B.2: The AXI-4 streams and their descriptions.

AXI-4 Stream	**Description**
TDATA	Data stream
TKEEP	Marks NULL bytes (i.e. byte enable)
TVALID	Validity indication
TREADY	Flow control indication
TLAST	End of packet/burst indication
TUSER	Sideband metadata

Appendix C

Reproducibility

Appendix C gives the specification of the test machine and the required software and equiment to reproduce the results presented in the dissertation.

C.1 Test Machine

The following machine, software and equipment were used:

- A PC with Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-83-generic x86_64).
- Xilinx Vivado 2018.2 (License)
- Xilinx P4-SDNet (License)
- Xilinx 10G MAC (License)
- The NetFPGA SUME board.

C.2 Scripts

The steps to reproduce the SDNet simulation, SUME simulation and hardware-based test are as followed:

1. Go to the repository:
   `cd P4-NetFPGA/`
2. Update the environment variables:
   `source tools/setting.sh`
3. To run both the SDNet and SUME simulation:
   `cd $P4_DESIGN_DIR && make workflow`
4. To run the SDNet simulation only:
   `cd $P4_DESIGN_DIR && make vivado_sim`
5. To run the SUME simulation only:
   `cd $P4_DESIGN_DIR && make sume_sim`
6. Once the SUME simulation finishes, to run the hardware test:
   `cd $SUME_FOLDER`

```
./tools/scripts/nf_test.py hw --major switch --minor default
```

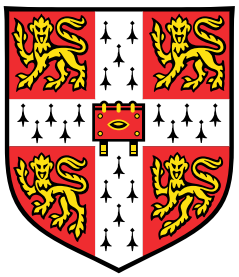
7. Once the hardware test finishes, to compile the bitstream for FPGA:
   ```
   cd $NF_DESIGN_DIR && make
   ```

Appendix D

Project Proposal

Computer Science Tripos - Part II Project Proposal

An accelerated, network-assisted TCP recovery



2385F

Supervisor: Dr Noa Zilberman

Department of Computer Science
University of Cambridge

Originator: Dr Noa Zilberman

Director of Studies: Dr Graeme Jenkinson

Project Overseers: Dr Markus Kuhn, Dr Eva Kalyvianaki and Dr Nada Amin

Downing College October 19, 2018

Introduction

Background

Transmission Control Protocol (TCP) is the protocol of choice in many data centers. However, it is very sensitive to losses (by design, as a mean for congestion control), which can degrade the performance within the data centers significantly [1]. Various congestion control and avoidance recovery mechanisms are thus of high importance in this field to minimise such loss rate. Still, not all TCP losses are born equal. For example, losses happening at the destination host's network interface card (NIC) are not an indication of congestion within the network. It is assumed that fast retransmission of such lost packets, from within the network, can increase the utilization of the network.

In-network computing is an emerging research area in systems and networking, where applications traditionally running on the host are offloaded to the network hardware (e.g., switch, NIC). Examples of applications offloaded in the past include network functions (DNS server [2]), distributed systems functions such as consensus (P4xos [3]), various caching (netCache [6], netChain [5]) and even a game (Tic-Tac-Toe). Key-Value Store (KVS) is also among the popular type of in-network applications.

Therefore, it is particularly interesting, and indeed challenging, to see how network-accelerated KVS concepts can be applied to TCP recovery in order to improve cross-datacentre performance.

The Project

Currently, when there is packet loss, signaled by duplicate ACKs (DUP ACKs), the DUP ACKs will traverse all the way back to the host (**Figure D.1**). The host receives the DUP ACKs, then resends the packet. The goal of my project is to implement a fast TCP recovery mechanism, which aims to reduce the response time to DUP ACKs and reduce changes to the congestion window by retransmitting packets from within the network (e.g., from the switch), instead of sending the DUP ACKs back to the host (**Figure D.2**). The implementation will be based on KVS concepts, where the key is the flow Id and sequence number, and the value is the packet.

For this project, I will be using the P4 programming language [19]. It is a language designed to allow the programming of packet forwarding planes. Besides, unlike general purpose languages such as C or Python, P4 is domain-specific with a number of constructs optimized around network data forwarding, hence is well suited to such a network application.

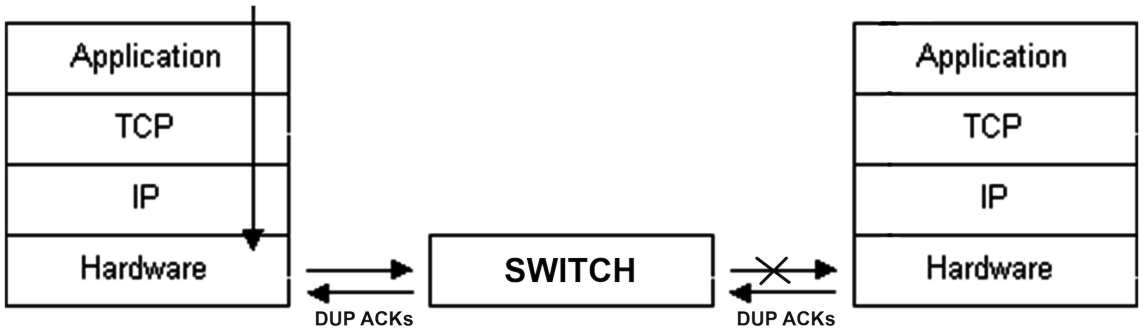


Figure D.1: The standard convention of TCP handling.

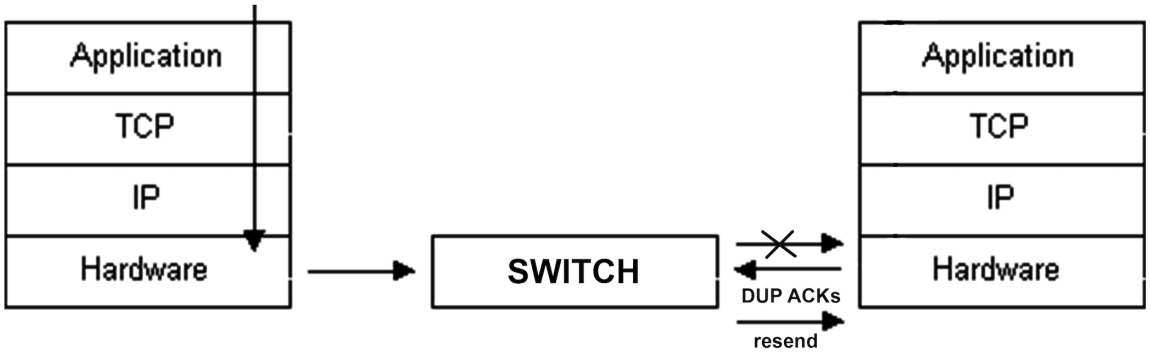


Figure D.2: The proposed TCP handling.

Starting Point

Platform & Language

This project will work mainly with a NetFPGA SUME board [24], using P4 programming language. I will be using the P4-NetFPGA workflow, which provides infrastructure to compile P4 programs to NetFPGA [32]. Apart from that, everything else will be built from scratch.

I have no prior experience with either NetFPGA or P4, but this will be mitigated through self-learning in which I will make use of the online tutorials, Google's resources and the P4 community documentation, as well as the experience of my supervisors.

Computer Science Tripos

The relevant Tripos courses that can serve as a starting point for this project are primarily: *Computer Networking*, *Principles of Communications* and *ECAD and Architecture Practical Classes*. Since the courses are introductory, I will also consult Part III's *High Performance Networking* course. I also plan to bridge any knowledge gap through extensive personal reading as well as help from my project supervisor.

Resources Required

For this project I will be using my own computer, a 2017 MacBook Pro with a 2.3 GHz Intel Core i5 processor and 16 GB of RAM, that runs macOS Mojave. I accept full responsiblity for this machine, and I have made contingency plans to deal with hardware and/or software failures. Should that machine suddenly fail, I have another 16Gb of RAM computer with a 2.6 GHz Intel Core i7 processor that runs Ubuntu 18.04 LTS. If all else fails, I can continue to work on an MCS machine. Backups will be done weekly to Microsoft OneDrive and my external hardrive, and all my codes will be uploaded to GitHub for version control.

For the hardware prototype, I will require a NetFPGA SUME board. I will need access to a machine that has the SUME installed, and a 10G NIC. These will be supplied by my supervisor. I will be using my own machine for development, with remote access to a server with the SUME board inside.

I will also require access to the lab network in order to *ssh* to the server with the SUME board. Development on my machine will require VPN access, for using floating licenses of the development tools. Lastly, I will require a second server for testing some of the extensions of the project.

Work to be Done

The main core component of the project is to be able to apply the KVS concept, where the sequence number and flow Id of a packet are the key and the packet is the value, to implement TCP fast recovery. In order to do that, the following sub-tasks must be done:

1. First, since the project is done on NetFPGA using P4 programming language, both of which I am not familar with, I first have to study them thoroughly and be proficient working with the platform.

2. I would also need to study and gain a better understanding of KVS applications, TCP congestion control and recovery mechanisms, as well as their use cases (e.g. High Frequency Trading, in-datacenter latency sensitive applications, etc.).

3. The next stage will be to design the architecture for the application. This is a non-trivial process. Generally, it will be to try to map the application to a match-action pipeline.

4. The fourth stage involves implementing the architecture in code. The program, which will be written in P4, will have the basic functionalities such as:

- A basic L3 switch function and TCP decoding (send TCP, read the flow information from the table/register)
- Matching packets to a "*key*" (its flow Id):
  - If the key is of a new packet, store it (SET()).
  - If the key is of a DUP ACK, read the packet and re-send (GET()).
  - If the number of DUP ACKs is greater than N, do not resend. Instead, forward to host as the standard convention of handling TCP.
  - Do not resend if the flow Id is not in a "selected" table. In other words, use the KVS only if the flow Id matches a predefined set of flow Ids, or a different predefined rule by the user.

5. **The simulation stage:** perform functionality test by simulating the program using bmv2 or Xilinx simulators to ensure correctness.
6. After it runs smoothly in the software, it will then be compiled to the hardware. Ensure that the basic functionalities mentioned above work in hardware.
7. **The hardware stage:** demonstrate interoperability with a software-based client/application.

   Possibly, I will implement a drop 1 : N packets at the server to force DUP ACKs, and I will also be using a synthetic benchmark such as *multilate* [41] – a memcached benchmark – or OSNT [40] as the applications on top of the network. The criterion to evaluate the performance is the flow completion time. There is no intention to use network simulators such as ns2, omnet++, etc. This is outside the scope of this project.

   The aim for this stage is to get a working prototype in hardware that supports a single flow and single packet size, and evaluate on its performance.
8. Once the prototype is up and working, I need to implement further extensions to allows the prototype to support a variety of parameters/conditions (which will be discussed in **Possible Extensions** section).

Success Criteria

This project will be deemed a success if I managed to study and design an architecture for the application, as well as succeeded to implement that design. More concretely:

1. I have an implementation of the application written in P4.

2. The implementation works in simulation (using bmv2/Xilinx simulators).

3. I have a working prototype. In other words, my design runs on the hardware.

4. I am able to demonstrate interoperability with a software-based client/application.

5. I can provide a performance evaluation of the design.

Possible Extensions

If the core parts of the project are successful and completed within a reasonable time, I shall then try to investigate and implement further extensions. Some possible options are:

1. Supporting more than a single flow, and supporting the configuration of flows to monitor (& retransmit)

2. Supporting different packet sizes (depends on workflow limitations).

3. Sending a notification to the source if multiple retransmits fail.

4. Adaptively installing and deleting from the list of flows to monitor.

5. A performance evaluation comparison to existing TCP recovery mechanisms.

Timetable

The schedule is broken into 15 2-week periods, with the first period starting on 19/10/2018.

1. **Michaelmas weeks 2–4 [19/10–5/11]:** Preparatory reading on the P4 programming language and setting up the NetFPGA platform. Going through the tutorials and experimenting with some examples. Study TCP congestion control and recovery mechanisms beyond the material taught in *Computer Networking* and *Principles of Communications* courses.

2. **Michaelmas weeks 5–6 [6/11–19/11]:** Design the architecture: mapping the application to a match-action pipeline.
   Milestone: Understand the application architecture. Be able to map the application to a match-action pipeline.

3. **Michaelmas weeks 7–8 [20/11–28/11]:** Start implementing of the design by

writing the basic code.

4. **Michaelmas vacation weeks 1–2 [30/11–12/12]:** Simulate the application using bmv2/Xilinx simulators.
   Milestone: **Basic design written in P4 working in simulation, with minimal bugs left.** .

5. **Michaelmas vacation weeks 3–4 [13/12–26/12]:** Start to implement the working code into hardware. Write a hardware-test program (Python-based).

6. **Michaelmas vacation weeks 5–7 [27/12–16/1]:** Demonstrate interoperability with a software-based client/application. Start writing the progress report.
   Milestone: **Have the application working in hardware, a completed progress report and a presentation for demonstration purposes.**

7. **Lent weeks 1–2 [17/1–30/1]:** Work on the design's performance testing and evaluation.

8. **Lent weeks 3–4 [31/1–13/2]:** Continue working on testing and evaluation of the design.
   Milestone: **The project is finished and meets the success criteria.**

9. **Lent weeks 5–6 [14/2–27/2]:** Start working on extensions, based on project's progress.
   Milestone: **The prototype continues to work well with the extensions implemented.**

10. **Lent weeks 7–8 [28/2–13/3]:** Possible overflow from the previous weeks. Clean up codes and repository. Start writing dissertation main chapters.
    Milestone: **Completed working prototype with core components and extensions in place. First draft of dissertation**

11. **Easter vacation:** Continue writing dissertation. Review cycles and corrections to dissertation towards the end of the vacation.

12. **Easter term 1–2 [25/4–8/5]:** Completed dissertation. Proof reading and then an early submission so as to concentrate on examination revision.

13. **Easter term 3 [9/5–17/5]:** Buffer week.