Thanh Bui
Downing College
ttb29

Computer Science Tripos - Part II Project

An accelerated, network-assisted TCP fast retransmit

May 17, 2019

Proforma

Name:	**Thanh Bui**
College:	**Downing College**
Project Title:	**An accelerated, network-assisted TCP fast retransmit**
Examination:	**Computer Science Tripos — Part II, June 2019**
Word Count:	**11000**[1]
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 a programmable switch to assist the TCP fast retransmit mechanism. The implementation will be evaluated based on a series of tests, including both software and hardware simulations. A performance evaluation will also be provided.

Work Completed

Almost all my success criteria were met, with the exception of demonstrating the interoperability of my implementation with a software-based client/application. The architecture was designed, then implemented in P4. The implementation was tested via three different simulations: SDNet simulation, SUME simulation and hardware simulation. A performance evaluation of the design was provided. Two of the extensions were also completed.

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

Special Difficulties

The design stage of the architecture took longer than anticipated due to the limitations of SDNet and the SimpleSumeSwitch architecture of the P4→NetFPGA workflow. The current P4→NetFPGA only supports programmable packet processing, i.e. operations on packet headers only, while this project used programmable buffering logic, which would require the ability to buffer packets. Hence, my first design cannot 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 framework that allow packet buffering. This rendered me unable to demonstrate the interoperability of my design with a software-based client/application and meet all my success criteria.

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.

SIGNED DATE

Contents

Chapter 1

Introduction

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

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 degrade the performance within the data centres significantly [1]. Various congestion control, avoidance and 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 utilisation 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 [4], 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 fast retransmit mechanism in order to improve cross-datacentre performance.

1.2 Project Aims

Fast retransmit is an enhancement to TCP that reduces the time a sender waits before retransmitting a lost segment. A TCP sender normally uses a simple timer to recognize lost segments. 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.

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 three subsequent packets are lost. The next packets the receiver sees are packet numbers 5 and 6. After receiving packet number 5, the receiver sends an acknowledgement, but still only for sequence number 2. When the receiver receives packet number 6, 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).

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 without waiting for its timeout.

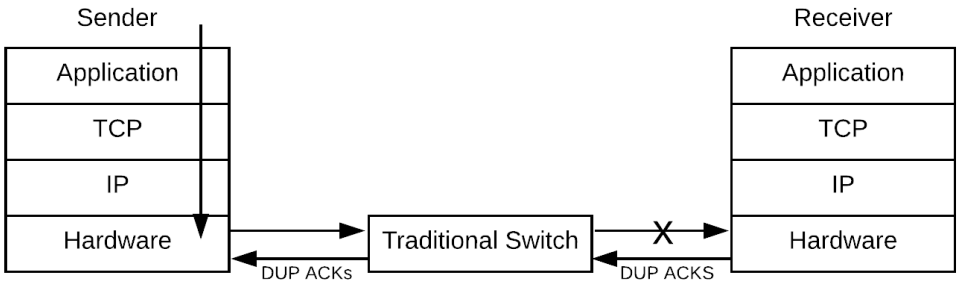

Figure 1.1: The standard convention of TCP fast retransmit.

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.

This project aims to design and implement a programmable switch that assists the TCP fast retransmit algorithm. The programmable switch will be able to retransmit

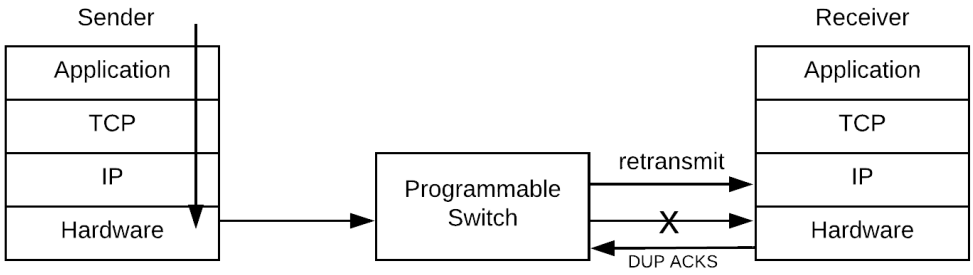

Figure 1.2: The proposed TCP fast retransmit, assisted by the programmable switch.

the packets from within the network, instead of waiting for the DUP ACKs to get back to the host (Figure 1.2), thereby aims to reduce the response time to DUP ACKs and reduce 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

One of the main aspects of TCP is congestion control, 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 are defined in more detail in RFC 5681[6]. In this project, we are mostly interested in the *fast retransmit* algorithm, which has been explained in the previous section.

1.3.2 Programmable Data Planes

In the last eight years, Software-defined Networking (SDN) and the OpenFlow protocol have reshaped the way people configure forwarding devices and determine network behaviour, by offering an open interface upon which apps like routing, monitoring,

etc. can be built. OpenFlow has been the *de facto* implementation of SDN. The key idea behind it was to 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. However, it is still very limited to a fixed set of features, which does not include new or custom protocols, statistics other than Packet, Byte count and Flow duration, and actions such as stateful matching or forwarding logic [7]. This can change drastically with the re-emergence of programmable data planes and languages like P4 [8–10] and Protocol Oblivious Forwarding (POF) [11, 12]. They enable faster development/provisioning of new and/or custom protocols, as opposed to the long wait for the release of fixed-function Application-Specific Integrated Circuit (ASIC) switches supporting standardised protocols [13]. Data plane programmability has the potential to unleash a new generation of 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 and the starting point for this project. I move on to present the formal requirements. This is followed by a discussion of the different components of a programmable data plane, including the P4 programming language, the P4-SDNet compiler, the NetFPGA platform and the P4→NetFPGA workflow. Finally, I discuss the project workflow.

2.1 Software Used

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

2.1.1 Programming Languages

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

- **P4** is a 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.

- **Python** was used extensively in the evaluation because of the `scapy` module, 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.

- **Verilog** was used to implement certain HDL modules within the P4-NetFPGA

platform, in order to add or modify certain functionalities to suit the purpose of my design. It is the language of choice of the P4-NetFPGA platform.

- **Tcl** was used to write project wrappers and debug scripts.

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

2.1.2 Development Tools

- **The NetFPGA SUME Board**[2] is an advanced board that features one of the largest and most complex FPGA's ever produced, a Xilinx Virtex-7 690T supporting thirty 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 environment to develop and test P4 programs using the Xilinx P4-SDNet[3] toolchain within the NetFPGA SUME reference switch design.

- **Vivado® Design Suite** is a software suite produced by Xilinx[4] for synthesis and analysis of HDL designs. Vivado was used in the project because it is the 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.

- **Backups** were taken by uploading relevant files to Microsoft OneDrive[5]. The git repository itself was hosted remotely on GitHub[6].

[2] A collaborative effort between Digilent, the University of Cambridge and Stanford University.
[3] https://www.xilinx.com/products/design-tools/software-zone/sdnet.html
[4] https://www.xilinx.com/
[5] https://onedrive.live.com/about/en-gb/
[6] https://github.com/ttbui11/part-ii-proj/

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 Part II and Part III courses:

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

In terms of familiarity, I had no prior experience with P4 programming language, the P4→NetFPGA workflow and Tcl, and little experience with Verilog, based on 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 main code in P4 and the tests in Python were written from scratch, using the template given by the P4→NetFPGA workflow as the starting point. The code for the additional modules and externs in Verilog as well as the project wrappers in Tcl are modified from some of the current modules to suit the required functionalities.

2.3 Requirements Analysis

This project has one software deliverable: an implementation of a programmable switch that will retransmit a packet when it receives the third DUP ACK from the receiver.

Below is a list of requirements and extensions for the deliverable, prioritised using *MoSCoW* criteria [14]:

Must have

- Have an implementation of the switch in P4.
- The implementation works correctly in an SDNet simulation.

- The implementation works correctly in a SUME simulation.
- The implementation works correctly in a hardware simulation.
- A performance evaluation of the design.

Should have

- The switch will send a notification to the source if the retransmit fails.
- A performance evaluation in comparison to existing TCP fast retransmit mechanism.

Could have

- The design will support more than a single flow, and support the configuration of flows to monitor.
- The design will support different packet sizes.
- 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

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 protocols like OpenFlow. This section gives a brief overview of the P4 programming language with the aim to provide sufficient basis of the project.

Many targets 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.

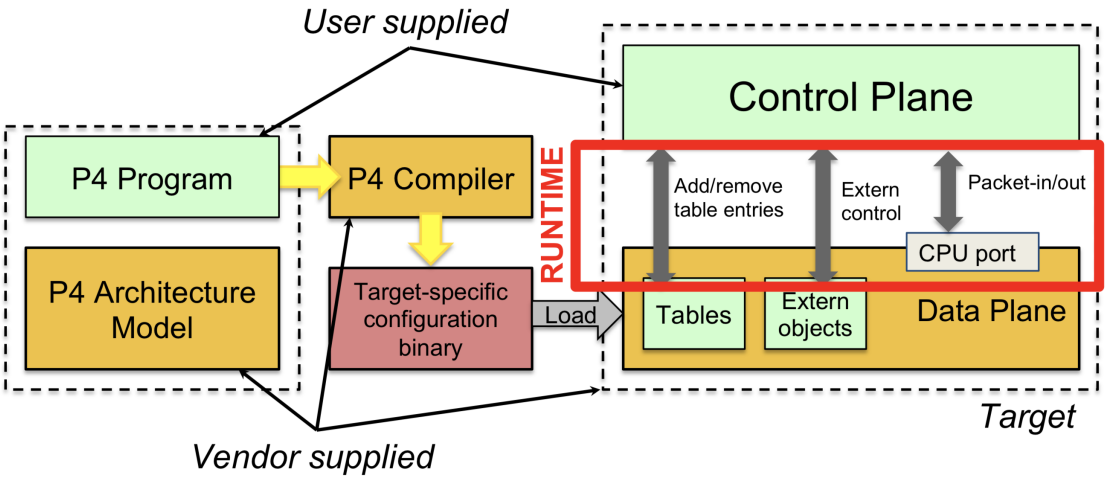

Figure 2.1: The process of programming a P4 target. Source: P4.org – Copyright © 2019.

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 P4-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 P4-SDNet

The Xilinx P4-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 [15], with a P4 to PX translator. Figure 2.2 depicts the process of compiling P4 programs that target the SimpleSumeSwitch architecture using P4-SDNet. The front end translator maps P4 programs into corresponding PX programs and also 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.

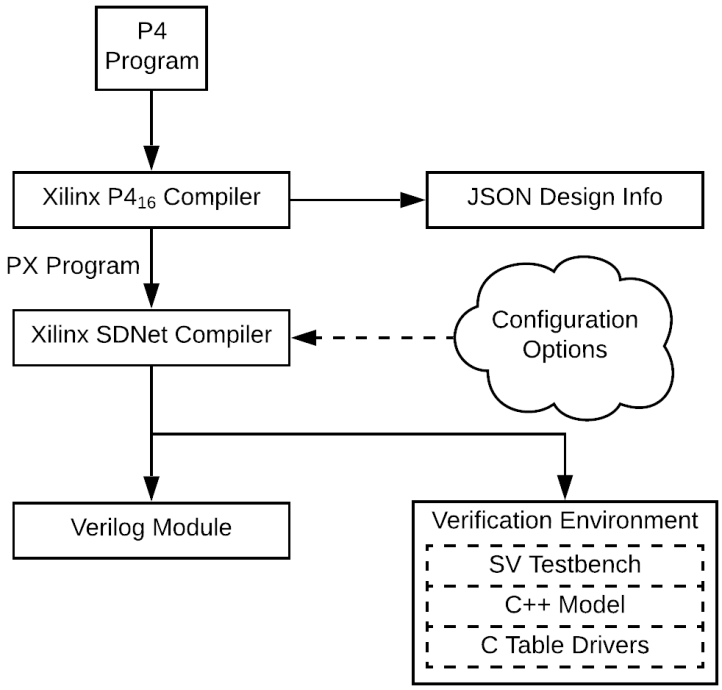

Figure 2.2: The Xilinx P4-SDNet compilation flow. P4 programs are first translated into a PX program, which is then compiled into a Verilog module using the SDNet flow. SDNet also produces a verification environment.

2.6 The NetFPGA Platform

The NetFPGA (Networked FPGA) project is a teaching and research tool designed to allow packets to be processed at line-rate in programmable hard-ware. 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 total I/O capacity of 100 Gb/s, 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 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.

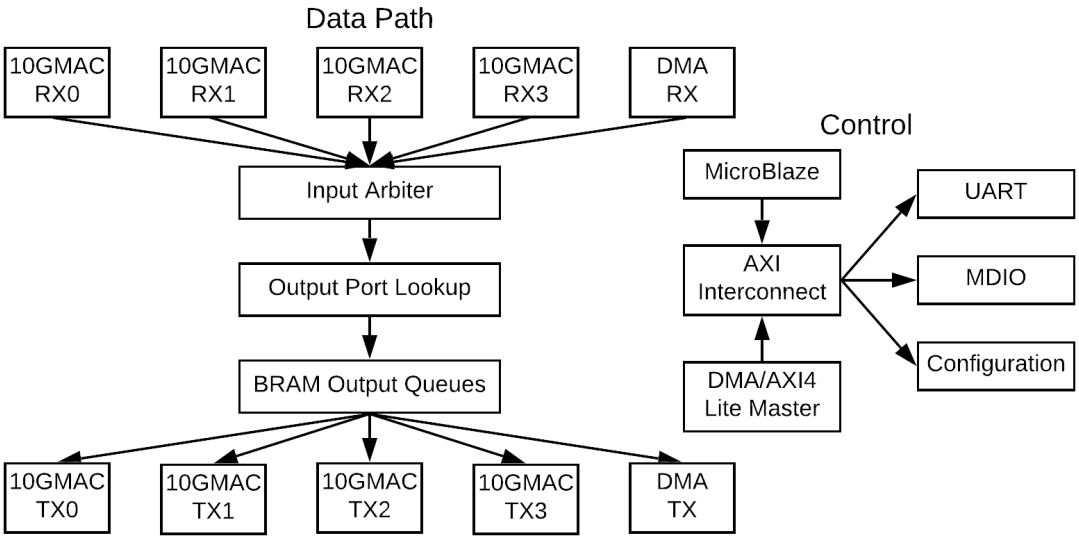

Figure 2.3: Block diagram of the NetFPGA reference design.

2.7 The P4→NetFPGA Workflow

The *SimpleSumeSwitch* (SSS) 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 2.4. The SSS is a great 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.

The SimpleSumeSwitch's `sume_metadata` bus corresponds to the tuser bus in the SUME reference switch design, and 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.

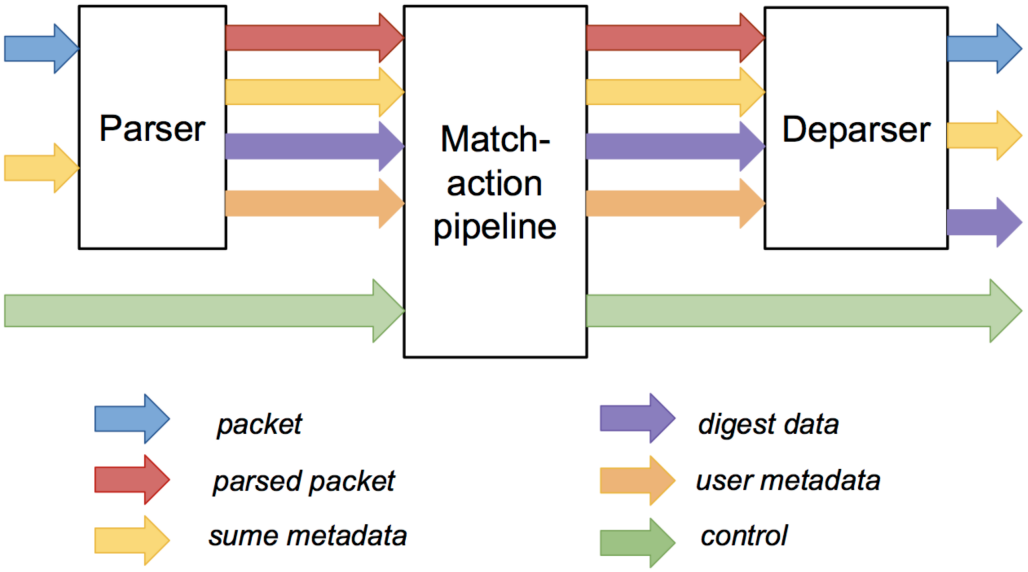

Figure 2.4: Block diagram of the SimpleSumeSwitch P4 architecture used within the P4→NetFPGA workflow. Source: P4→NetFPGA Home Wiki.

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

Figure 2.5 outlines the automated P4→NetFPGA workflow. We first write a P4 program

which is compiled (by Xilinx P4-SDNet) into an HDL instance of the SSS architecture. The SSS module is then automatically integrated into the NetFPGA reference switch design by replacing the default Output Port Lookup module. The flexibility of the SSS architecture means that it could also 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. Section 2.8.2 will explain the customised architecture in more detail.

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.

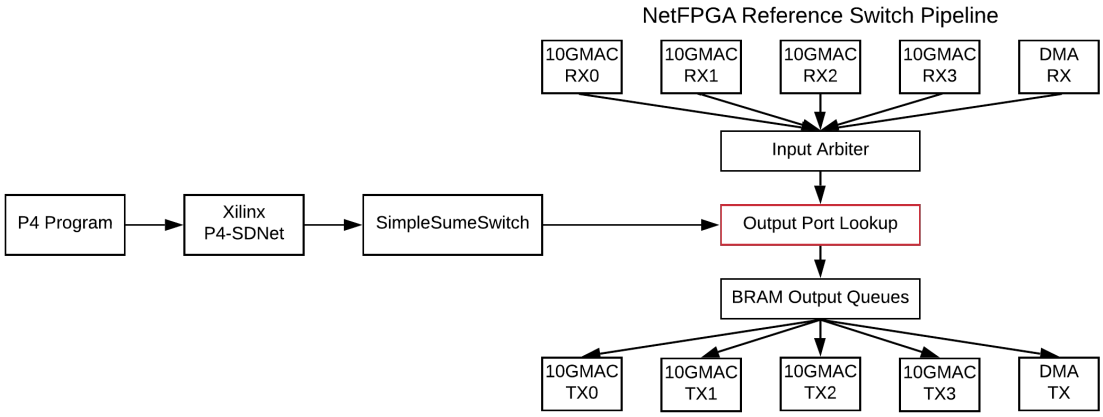

Figure 2.5: The automated P4→NetFPGA compilation flow. P4 programs are compiled into an HDL instance of the SimpleSumeSwitch architecture, which is then used to replace the Output Port Lookup module in the NetFPGA Reference Switch Design.

2.8 Project Workflow

2.8.1 Preparation Stage

I spent the first two weeks of the project learning the P4 language, setting up the development environment and learning the P4→NetFPGA workflow.

The P4 Language Consortium [10] provides a set of exercises to get me started. I completed their tutorial[7], 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[8]. The workflow provides a template for a general P4 program following the SSS architecture model, from which I will start to write my implementation, as stated in Section 2.2. The main challenge of this part is learning Verilog and Tcl in a short amount of time.

2.8.2 Architecture Design Stage

Following the preparation stage, I designed the architecture of the switch. I would then write P4 programs that follow the architecture and make adjustments to the design where necessary.

[7] https://github.com/p4lang/tutorials
[8] https://github.com/NetFPGA

Network Level

The switch will embed the TCP fast retransmit mechanism and will be one hop away from the receiver, as illustrated in Figure 2.6. Since the switch is programmable, it will be configured to buffer only packets of particular applications, identified by their flow number, while transmitting other packets normally. This design avoids introducing unnecessary buffering of packets of other flows which is the cause of bufferbloat.

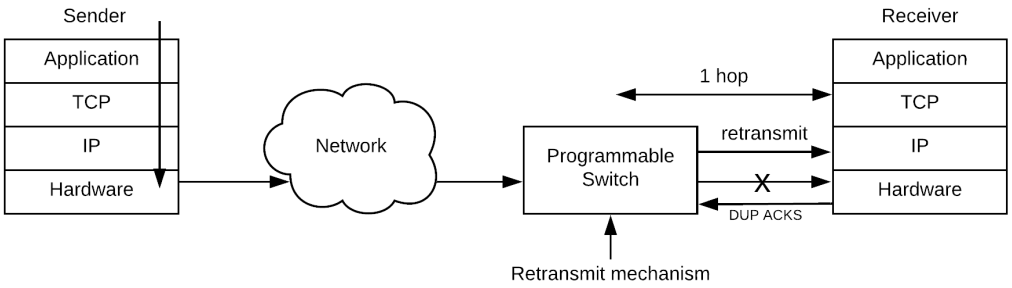

Figure 2.6: The network.

System Level

The first design uses a hash table to map flow number to another hash table with the packet sequence number as the key and the packet itself as the value. The flowchart showing the buffering logic of this design is given in Figure 2.7. However, this design does not work due to the limitations of SDNet and the SSS architecture of the NetFPGA platform. The SSS architecture only supports programmable packet processing, 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.

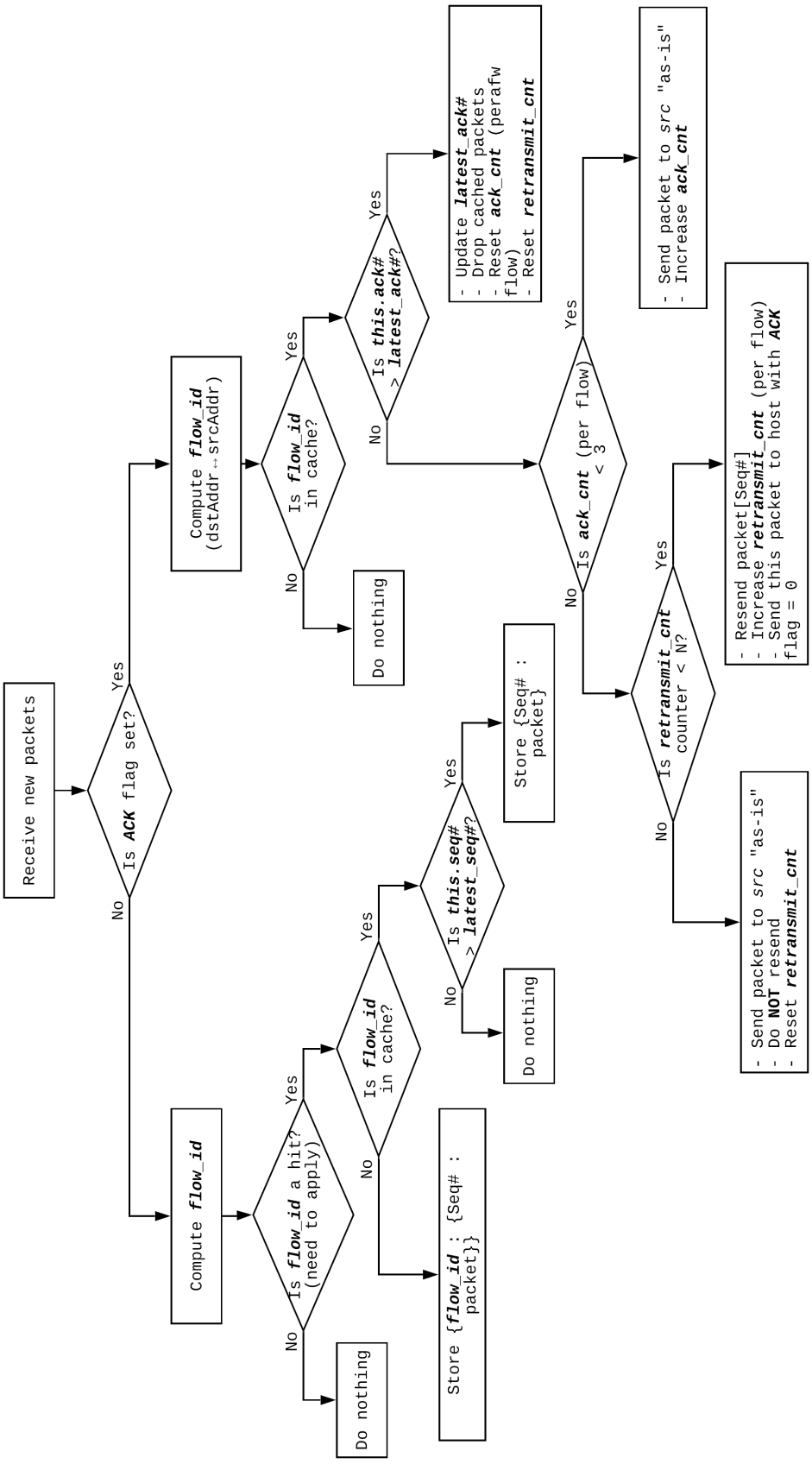

Figure 2.7: Flowchart shows the packet buffering logic of the first design.

After careful study of the first design and the platform, I decided to add an additional HDL module that follows the SSS module in the NetFPGA reference switch pipeline. The additional HDL module is called the Cache Queue and it will buffer packets to be retransmitted. Our reference switch pipeline will now look like Figure 2.9. In this pipeline, when a packet exits the SSS 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.

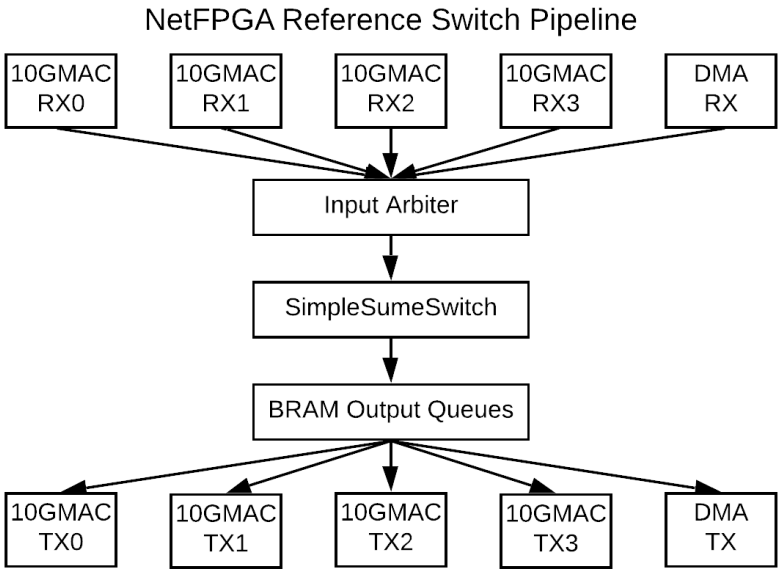

Figure 2.8: Block diagram of the NetFPGA reference switch design.

2.8.3 Implementation Stage

In order to fulfil the requirements analysis, I follow the *spiral development model* [16] 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. Figure 2.10 shows the workflow of the implementation stage of this project. After laying out the requirements and designing the architecture, I will start to write the implementation in 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

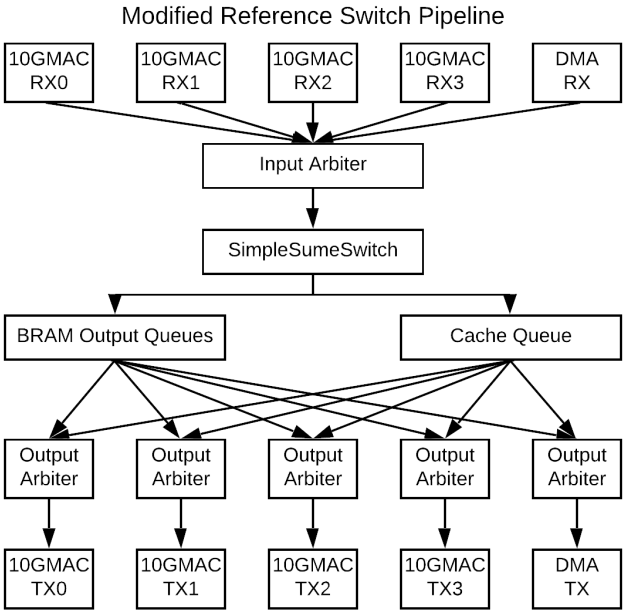

Figure 2.9: Block diagram of the modified reference switch pipeline. Packets are duplicated after the SSS module and being buffered in the Cache Queue.

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.

2.8.4 Risk Analysis

The P4→NetFPGA workflow is a complex platform that required the knowledge of a multitude of languages, with limited documentation [17] and community support [18]. 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 [16], this high-risk part was scheduled early and some "catch-up" time was allocated in the project timetable in case it caused significant delays.

2.8.5 Backup Plan

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

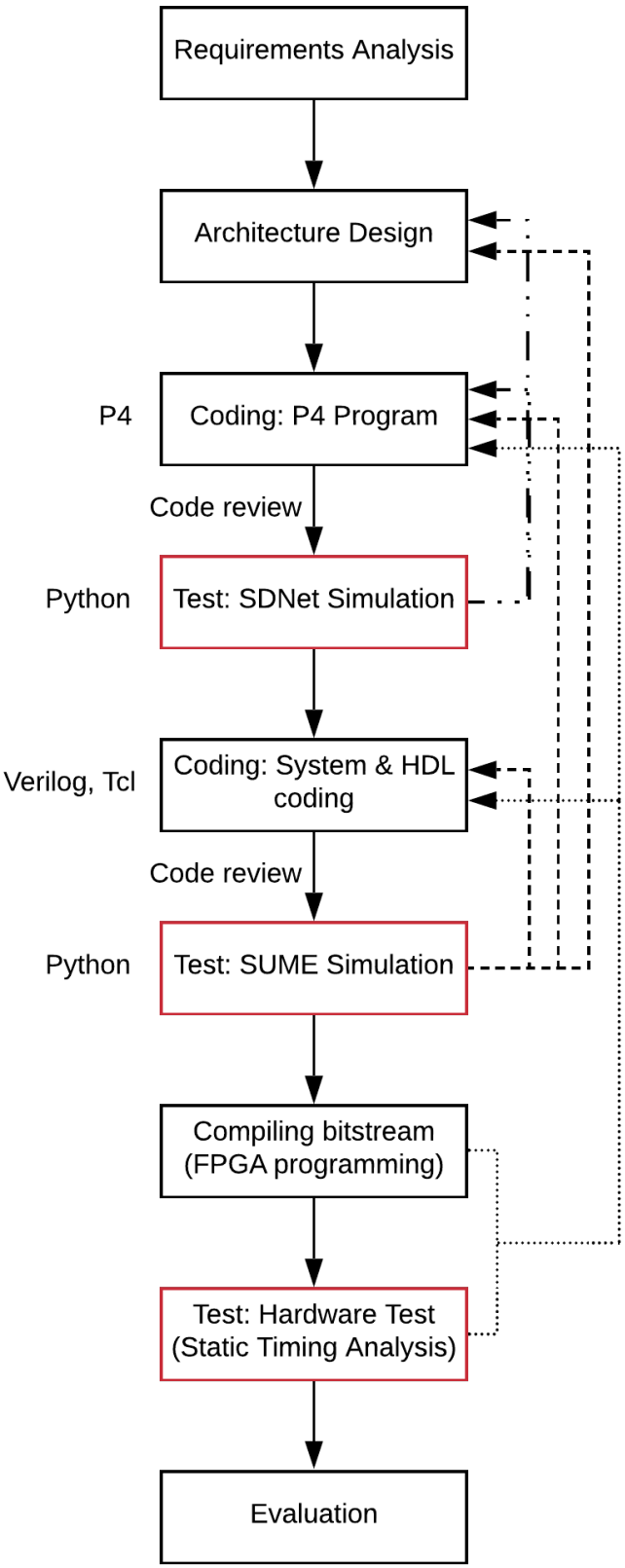

Figure 2.10: Block diagram showing the workflow of the implementation stage. Dotted arrows represent a revision of previous steps, possibly with adjustments/refinements, in an iterative approach. Where appropriate, the programming language involved is stated. Passing all the steps in red box indicates the design meeting the requirements.

Chapter 3

Implementation

I give the overview of the project repository. I move on to explain the

3.1 Repository Overview

```
P4-NetFPGA
├── project
│   ├── simple_sume_switch
│   │   ├── hw
│   │   │   └── hdl
│   │   │       └── nf_datapath.v*
│   │   └── 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*
```

This project will work mainly with a NetFPGA SUME board [19], using P4 programming

language. I will be using the P4-NetFPGA workflow, which provides infrastructure to compile P4 programs to NetFPGA [17]. Apart from that, everything else will be built from scratch.

3.2 Software Implementation

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. The following code sequence defines our parser with a `start` state:

```
// Parser Implementation
@Xilinx_MaxPacketRegion(8192)
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) {
  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;
    }
  }
}
```

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 3.1 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

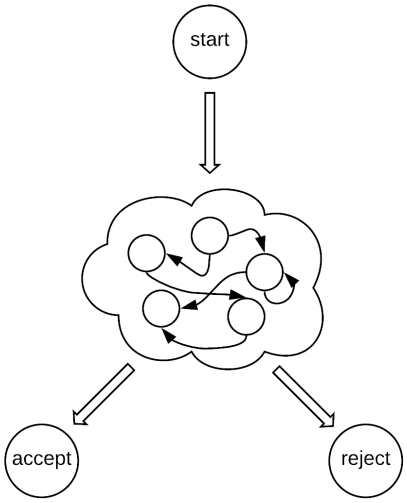

Figure 3.1: Parser general FSM structure.

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 adopts the former.

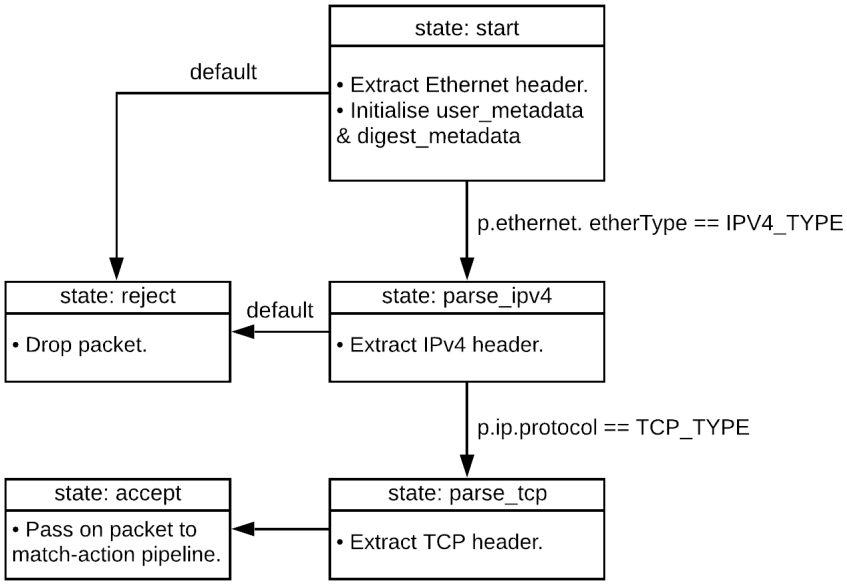

Figure 3.2: The FSM of the design.

Figure 3.2 describes the FSM structure of our parser, in which I defined two additional states `parse_ipv4` and `parse_tcp`:

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

Figure 3.3: The definition of `parse_ipv4` state.

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

Figure 3.4: The definition of `parse_tcp` state.

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.

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. This match-action processing model was originally introduced as the core around which the OpenFlow model for SDN was built []. 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 was not used in this design.

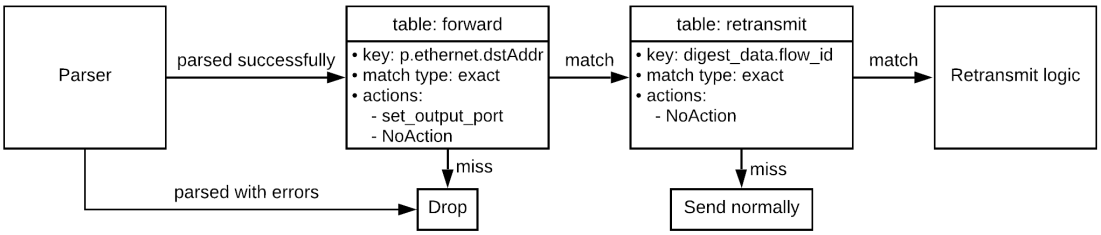

Figure 3.5: A packet processing program describing a simple L2/L3 IPv4 switch.

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. Table entries are populated at runtime by the control plane software.

Table 3.1: Entries for the `forward` table.

Key	Action ID	Action Data
1	16	1
1	8	1
1	8	1
1	8	1
1	16	1

Table 3.2: Entries for the `retransmit` table.

Key	Action ID	Action Data
1	16	1
1	8	1
1	8	1
1	8	1
1	16	1

Figure 3.5 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 first table uses the Ethernet destination address to determine the output port for the next hop. If this lookup fails, the packet is dropped (by assigning `sume_metadata.dst_port` to 0). The second table checks the computed `digest_data.flow_id`: if it matches our flow of interest, the packet will be monitored to assist the fast retransmit of TCP congestion control. Otherwise, the packet will just be sent normally. A packet will be modified by a series of `actions`: `set_output_port` sets the output port on the SUME board for packets whose Ethernet destination address matches what was defined in the first table, and `compute_flow_id` computes the flow number of the packet for the second table to look up. `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.

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

- Receive and parse packet from the sender.
- Look up the Ethernet destination address to determine the output port. Drop on a miss.
- Compute the flow number of the packet.
- Look up the flow number of the packet to determine if it should be monitored. Send normally on a miss.
- Set `digest_data.tuser` and/or set the `ACK` flag in the TCP header appropriately.
- Construct the final packet and send it to the receiver.

3.2.3 The Extern Functions

P4 extern functions, or externs, are platform-specific functions that are not described in the core P4 language—a kind of "black boxes" for P4 programs. Extern functions are implemented in HDL and 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 [20]. P4→NetFPGA provides a set of commonly-used extern functions, shown in Table 3.3, in the `templates` folder.

Table 3.3: The P4→NetFPGA extern functions library.

Stateful Atomic Extern Functions	
Name	**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	
Name	**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)

An extern function can be declared using the syntax `extern void <name>_<extern_type>(in T1 in_da`
The following code sequence shows an example of declaring a afafa, dafaf:

```
@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.

For this design, I defined the following externs using the available extern functions provided by p4→NetFPGA:

-
-

MINUS Functionality

An 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 other words, each stateful atom 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. This complicates my design since WE REQUIRE INTUITIVELY TO WRITE AND CHECK.

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. 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. 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);
    }
}
```

3.3 Hardware Implementation

3.3.1 The nf_datapath – Some better name

3.3.2 The Cache Queue

3.3.3 The Output Arbiter

All input interfaces share the same bandwidth (and therefore width) as the output stream to ensure that maximum throughput can be achieved. The input port buffering will be handled in the AXI Converter block.admits packets from the ports into the data path

The function of this block is to merge a number of input streams into one output stream. All input interfaces share the same bandwidth (and therefore width) as the output stream to ensure that maximum throughput can be achieved. The input port buffering will be handled in the AXI Converter block. The P4→NetFPGA platform comes with an input arbiter, which performs the following functions:

- It receives packets from one of the physical input Ethernet ports, from the control plane, or from the input recirculation port.
- For packets received from Ethernet ports, the block computes the Ethernet trailer checksum and verifies it. If the checksum does not match, the packet is discarded. If the checksum does match, it is removed from the packet payload.

- Receiving a packet involves running an arbitration algorithm if multiple packets are available.

- If the arbiter block is busy processing a previous packet and no queue space is available, input ports may drop arriving packets, without indicating the fact that the packets were dropped in any way.

- After receiving a packet, the arbiter block sets the inCtrl.inputPort value that is an input to the match-action pipeline with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0 to 7, while the input recirculation port has a number 13 and the CPU port has the number 14.

For our architecture, since we have two queues—the output queue and the cache queue—we would require an "output" arbiter for each of the five ports.

Chapter 4

Evaluation

This chapter assesses my implementation functionality with regard to the initial requirements across three different environments. I then evaluate the performance of my design. Finally, I compare the performance of my design to existing TCP fast retransmit mechanism.

4.1 SDNet Simulation

4.2 SUME Simulation

4.3 Hardware Test

4.4 Performance Evaluation

4.5 Performance Evaluation

Chapter 5

Conclusion

5.1 Accomplishments

Overall, the project achieved its aim of implementing and evaluating a programmable switch that is capable of assisting the fast retransmit process of TCP. The switch functionalities were evaluated at three different stages: a software simulation, a SUME simulation and a hardware test. A

An evaluation for the switch was also provided as comparing to TCP

With the benefit of hindsight, I would have implemented the architecture prior to starting the project and used it as the starting point. This would have enabled me to focus more on evaluating and give more time to explore useful extensions.

5.2 Future Work

Many promising avenues for further improvement were not explored due to time constraints:

- **Supporting multiple packet sizes**. The current design only supports a single packet size. It would definitely be more useful if the design could support a
- **Supporting multiple flows**. Embedding schemes that withstand transcoding would be useful for things like communication over social media. Services such as YouTube, Facebook and Tumblr transcode user-uploaded videos before presenting them to other users, destroying any payload hidden in motion vectors. Embedding algorithms that make use of error correcting codes or redundant encoding may be better able to resist this.
- **User INteface**. The current CLI could be replaced by a more novice-friendly

GUI, providing helpful guidance for inexperienced users who want to start using covert communication. This could allow

adaptively installing and removing flows to monitor, as well as the ability to configure the flow.It could include a privacy advice and an automatic algorithm recommendation based on a set of predefined use-cases, the payload size and the video embedding capacity. An HCI trial could be conducted to evaluate whether the interface helps the users achieve the required communication secrecy level while being educative and easy to use.

Closing Remarks

This project has been a fascinating opportunity to explore the field of computer networking, especially high-performance networking, comprehend and appreciate the intricacy of TCP congestion control mechanism and the potential of data plane programmability. The project has achieved most of its goals and attempted to investigate and evaluate a modification to assist TCP fast retransmit algorithm, providing a starting point for future improvements. This project has also contributed to my personal development by improving my software engineering and technical writing skills.

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] 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

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

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

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