Modelling and Performance Analysis of OpenFlow Switches in Software-Defined Networking

by

Deepak Kumar Singh

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Network Engineering.

Victoria University of Wellington
2019

Abstract

Software-Defined-Networking (SDN) simplifies the configuration complexity in the computer communication network by decoupling the control plane from the data plane in a switch. In SDN, the switch has the data plane only and is configured by the logically centralised controller which simplifies the forwarding of packets in the network. However, an SDN switch is sensitive to delay and loss of packets which significantly affects the network performance.

This thesis uses queueing theory to conduct modelling and performance analysis of OpenFlow-based SDN switches. OpenFlow is the defacto protocol for communication between an SDN switch and the controller. Using queueing theory, three aspects of packet processing in an SDN switch are explored. First, the existing research has primarily modelled the output buffer of an SDN switch using two buffer sharing mechanisms: the single shared buffer and the priority buffer. However, the effect of buffer dimensioning in these buffer sharing mechanisms has not been investigated. Buffer dimensioning helps in determining the minimum buffer capacity for a desired loss probability. The research in this thesis shows that the use of priority buffer in an SDN switch reduces the time to update flow tables than the shared buffer but at the cost of a higher buffer capacity.

Second, much of the existing research has not investigated the impact of internal buffering of data packets whereby a fraction of a data packet header is sent to the controller instead of an entire data packet. To investigate the impact of internal buffering, the queueing model for an SDN switch with the internal buffer is developed. The investigation shows that

at the time of congestion, the internal buffer in an SDN switch improves the network performance with lower delay and lower packet loss.

Finally, existing research has focused on a software switch in SDN and very little research has studied the performance of a hardware switch. To characterise the performance of SDN-based hardware and software switches and identify the tradeoffs between them, a unified queueing model has been developed. The unified queueing model is an analytical tool for network engineers to predict delay and packet loss in their SDN deployments. The analysis shows the benefits of a hardware switch over a software switch. These benefits are lower delay and lower packet loss. However, the increasing involvement of the controller reduces the benefit of using a hardware switch, i.e. forwarding packets at the line speed rate.

This research guides network designers and analysts in the selection of the shared or buffer model for an SDN switch for their desired Quality of Service (QoS). Furthermore, the developed queueing model for an SDN switch with the internal buffer studies the impact of internal buffering in an SDN switch. Finally, the unified queueing model helps in the selection of a software or hardware switch in SDN.

Acknowledgments

I would like to express my deep gratitude to my supervisors Dr. Bryan Ng and Professor Winston Seah for their exemplary guidance, monitoring and constant encouragement throughout the progress of my thesis. I would like to extend my gratitude to Professor Yuan-Chen Lai and Professor Ying-Dar Lin for their informative suggestions and motivation for my hard work.

I also take this opportunity to express my gratitude to Dr. Ying Qu, Dr. Hang Yu, Dr. Liang Yang, Dr. Adrian Pekar, Kirsten Reid, Jakob Pfender, and Jordan Ansell for their cordial support, valuable information and guidance, which helped me in my research activities through various stages.

I am obliged to my colleagues for their cooperation during the period of my stay. Lastly, I thank almighty, my parents, sister Annu, and brother-in-law Nikhil for their constant encouragement throughout my study.

Publications

- **Deepak Singh**, Bryan Ng, Y.C. Lai, Y.D. Lin and Winston K.G. Seah, 'Modelling Software-Defined Networking: Switch Design with Finite Buffer and Priority Queueing', *Proceedings of the 42nd IEEE Conference on Local Computer Networks* (LCN), October 9-12, 2017, Singapore.
- **Deepak Singh**, Bryan Ng, Y.C. Lai, Y.D. Lin and Winston K.G. Seah, 'Modelling Switches with Internal Buffering in Software-Defined Networks', *Proceedings of the 27th International Conference on Computer Communication and Networks* (ICCCN 2018), July 30 -August 2, 2018, Hangzhou, China.
- **Deepak Singh**, Bryan Ng, Y.C. Lai, Y.D. Lin and Winston K.G. Seah, Modelling Software-Defined Networking: Software and Hardware Switches, *Journal of Networking and Computer Applications*, Vol 122, 15 Nov 2018.
- **Deepak Singh**, Bryan Ng, Y.C. Lai, Y.D. Lin and Winston K.G. Seah, Analytical modelling of Software and Hardware Switches with Internal Buffer in Software-Defined Networks, accepted by the *Journal of Networking and Computer Applications*, 18 March 2019.
- **Deepak Singh**, Bryan Ng, Y.C. Lai, Y.D. Lin and Winston K.G. Seah, Encapsulation vs. Internal Buffering: Performance Analysis of OpenFlow-Based Hardware Switch, *submitted* to *IEEE Transactions on Network and Service Management*, 17 Feb 2019.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A switch/router in a computer communication network is a networking device that primarily forwards packets. In traditional networking, a switch/router has a control plane and data plane coupled together [2]. The control plane performs control functions such as routing protocols and middle-box configuration while the data plane forwards packets based on decisions made by the control plane. This makes the control function hardware-dependent in the traditional network resulting in a complex network configuration. In addition, the use of a low-level programming language and vendor dependency makes it difficult for network operators during configuration. To simplify the configuration complexity in the traditional network, a new networking paradigm called Software-Defined Networking (SDN) that uses a high-level programming language has emerged [3].

SDN is a network paradigm in which the control plane is decoupled from a switch/router. In an SDN, the control function is moved to a logically centralised controller which eases the load on a switch/router with the primary function of forwarding packets. It allows the network to be controlled and managed through a software written in a high-level programming language. The shift of control function from a hardware to software dependency allows the system to be more agile, programmable and centrally managed. This makes SDN a promising approach for future networks and services such as Software-Defined Wireless Access Network (SDWAN) [5], 5G mobile networks [6], Industrial Internet of Things (IIoT) [7], Software-Defined Information Centric Network (ICN) [8], and Software-Defined Smart Home [9]. The concept of an SDN is realised with OpenFlow which is among the first and most widely used specification to define communication between the controller and switch in an SDN paradigm [4].

SDN is a promising network paradigm which overcomes limitations of the traditional network [10]. However, research into identifying the potential bottlenecks of an SDN is still being conducted to improve the overall performance of an SDN. The performance issues for packet processing in an SDN switch that require further study and investigation are buffer sharing mechanisms, packet encapsulation methods, and SDN switch types. There are various buffer sharing models which need to be compared and analysed to address design challenges such as provisioning of buffer capacity and protection of control packets in an SDN switch [11, 12]. Similarly, there are provisions in OpenFlow for various packet encapsulation methods at an SDN switch which are still unexplored [13]. These encapsulation methods for processing data packets at SDN switch need to be studied and analysed to provide insights and guidelines to network analysts. Finally, SDN switches are generally categorised into software switches and hardware switches [14]. These switches need to be compared to develop guidelines that can be used during SDN deployment.

The performance analysis of an SDN has been done by simulations, measurements or experimental testbeds, and analytical models [2, 15]. While simulations and experimental testbeds have their own advantages, analytical models can be used to verify and authenticate simulated or experimental results [16, 17]. Analytical modelling of an SDN switch provides important insights for benchmarking switch performance and parametric sensitivity analysis to help network engineers identify critical factors that may influence network performance.

This thesis contributes to analytical modelling and performance analysis of a reactive SDN architecture using queueing theory. It investigates different aspects of packet processing in an SDN switch. The outcomes for these aspects will help network designers and analysts to achieve the desirable network performance.

1.1 Motivation

Analytical models of an SDN switch are a key step for SDN performance characterisation. These models will help network engineers design networks suitable for delay and packet loss sensitive applications like industrial automation systems, interactive video, network gaming, VoIP, and online surgery [18, 19, 20, 21]. Analytical models can be used in validating experimental or simulated results with the theoretical support in the form of mathematical expressions, theorems and proofs. They have the advantage of being a cheaper and faster approaches for performance analysis than experimental testbeds and simulations.

In an SDN switch, the control traffic feedback by the controller carries important flow updates. The choice of buffer sharing mechanisms for an SDN switch for the protection of flow updates has a significant impact on the overall performance of an SDN [11, 12]. Therefore, the selection of an appropriate buffer sharing mechanism that can provide protection to flow updates and accurately capture SDN behaviour need to be studied. Generally, there are two buffer sharing mechanisms used for an SDN switch which are: *shared buffer model* and *priority buffer model*. The shared buffer model is the usual assumption for an SDN switch with a single shared buffer for both control traffic and external data traffic arriving at the switch. On the other hand, the priority buffer model uses a high priority class for the control traffic and low priority class for the data traffic [11]. Clearly, the priority buffer model provides protection to the flow updates but its effect on the overall performance of an SDN remains unclear. A comparative analysis between the shared and priority buffer models will guide switch designers in their SDN deployments.

Similarly, the data packet in a flow that arrives at the switch may not have matching flow table entries is sent to the controller. Based on the OpenFlow specification [13], the data packet seeking a flow update from the controller can be processed via three packet encapsulation methods: *full encapsulation*, *internal buffering*, and *partial encapsulation*. With the full encapsulation method, the entire data packet is encapsulated and sent to the controller. Whereas in the internal buffering method, the data packet is temporarily buffered and a fraction of the data packet is only encapsulated. In the partial encapsulation method, if there is sufficient memory to buffer packets, the internal buffering method is used otherwise the full encapsulation method is used. The full encapsulation method is the default packet processing method and is widely used for SDN switches [1]. Consequently, the remaining encapsulation methods, the internal buffering and partial encapsulation methods have not been explored much. The selection of a packet encapsulation method at an SDN switch also affects the overall performance of an SDN that need to be studied and investigated. Therefore, the comparative analysis between these packet encapsulation methods will guide network analysts to obtain the best performance during their SDN deployments.

Lastly, unlike traditional networking, an SDN switch can also be categorised as a software or hardware switch. A software-based SDN switch is a virtual switch such that is instantiated in a virtual machine, whereas a hardware-based SDN switch is a physical switch [4]. A hardware switch processes the incoming packets at the line speed rate but is resource constraint due to the cost, size and energy consumption. On the other hand, a software switch is cheaper, consumes less power, is scalable and offers flexibility to implement complex actions but at the cost of slower performance [14]. Both software and hardware switches have strengths and weaknesses. Therefore, more research needs to be conducted into why one could be selected rather than other to achieve an optimal performance in an SDN.

Through research in analytical modelling using queueing theory, realistic queueing models for an SDN are expected that will represent SDN behaviour accurately and help in understanding various aspects of an SDN switch through performance analyses. These models will guide switch designers and network analysts in their SDN deployments.

1.2 Research Challenges

Modelling and performance analysis of an SDN switch come with the challenges that are attributed to the characteristics of an SDN and limitations of analytical models. The research challenges associated with the modelling and performance analysis of an SDN switch in this thesis are listed below:

- **Estimation of a cost-effective buffer capacity for an SDN switch.** Buffer dimensioning helps in estimating the minimum buffer capacity for the switch such that packet loss in the switch is no more than a desired link loss rate. However, buffer dimensioning estimates the buffer capacity based on parameters such as the arrival rate, the service rate, and the server utilisation. The estimated buffer capacity may not be cost-effective and feasible for the desired loss probability. Therefore, buffer dimensioning for different buffer sharing mechanisms should consider the trade-off between Quality of Service (QoS) and the cost factor.

- **Modelling of an SDN switch with the internal buffer.** With internal buffering in an SDN switch, the forwarding delay of packets can be decreased [106], Quality of Service can be improved with reduced packet loss [122], and bandwidth of the control channel can be optimised [1]. However, internal buffering requires sufficient memory which is not feasible in low-end SDN switches. Therefore, modelling of an SDN switch with the internal buffer needs to assume that the switch has sufficient memory to characterise internal buffering.

- **Modelling of a hardware-based SDN switch.** A hardware switch maintains a software flow table in the central processing unit (CPU) and a hardware flow table in the specialised hardware for forwarding of packets. The software and hardware flow tables are synchronised through a middleware layer on the switch to avoid duplicate entries and to ensure consistent forwarding behaviour. However, it is difficult to characterise the synchronisation between software and hardware flow tables in a hardware switch. Therefore, modelling of a hardware-based SDN switch requires assumption that the CPU synchronises the flow tables with the specialised hardware.

1.3 Research Goals

The associated research goals are as follows:

- **To provide guidelines to switch designers that include selection criteria for different buffering strategies and the minimum buffer capacity for the desired loss probability.** This goal aims at modelling the output buffer of an SDN switch using queueing theory where external and internal traffic arrives at the switch. The two main objectives of this goal are: (i) to compare existing buffer sharing models and (ii) to perform buffer dimensioning for informing switch design.

- **To provide insights to network analysts for the use of internal buffering in an SDN with the prediction of performance measures**

such as the delay and packet loss. This goal aims to use queueing theory to derive a first-order estimate of an SDN switch performance and to identify potential trade-offs between switch designs with and without the internal buffer. This goal is to develop the queueing model to study internal buffering in an SDN switch. The two main objectives of this goal are (a) to investigate the effect of the internal buffer in the performance of an SDN switch with the help of a queueing model, and (b) to compare the queueing models for an SDN switch with and without internal buffer, and hence identify the trade-offs.

- **To provide guidelines to network engineers for deployment choices such as under what operating conditions a software data plane outperforms a hardware data plane and vice versa.** The final goal is to develop the unified queueing model with software and hardware switches. The unified queueing model will act as an analytical tool for engineers to the predict delay, packet loss and throughput in their SDN deployments. This goal has two main objectives, (i) to develop queueing models for a hardware-based SDN switches, and (ii) to compare queueing models for a software-based and hardware-based SDN switch.

1.4 Contributions

This thesis contributes to the analytical modelling of an SDN paradigm in the context of performance analysis for an SDN switch. The contributions are listed as follows.

- **Shared buffer vs. Priority queueing buffer.** Queueing models for the shared buffer model and the priority queueing buffer model for an SDN switch were compared. Based on the comparative analysis, the priority queueing model has been shown to provide shorter delay while installing flow table entries in the switch. However, this benefit comes at the cost of a higher buffer capacity required for the priority buffer than the shared buffer model. Adopting a priority queue for an SDN switch provides better isolation between the data and control traffic.

- **Internal Buffering.** The queueing model for an SDN switch with an internal buffer was developed to investigate the potential benefits and trade-off of internal buffering in SDN switches. From the comparative analysis with an SDN switch without an internal buffer, it was observed that a switch with an internal buffer reduced the average delay and packet loss at the cost of extra switch buffer capacity required. Therefore, the internal buffering in an SDN switch significantly reduces the average delay and loss probability at the time of contention.

- **Software vs. Hardware switch.** Unified queueing models with and without the internal buffer were developed to characterise the performance of SDN-based hardware and software switches. SDN-based hardware switches with and without the internal buffer have not been investigated much, especially from the analytical modelling aspect. Therefore, unified queueing models are useful tools for network analysts to get quick insights into both SDN-based software and hardware switches. These tools help network analysts to predict performance measures such as delay and loss probability in SDN-based software and hardware switches. Delay and loss probability are important metrics for delay and loss sensitive applications in computer networks.

1.5 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 provides the overview of an SDN paradigm with different aspects of packet processing in an SDN switch. The related works on performance analysis of an SDN switch using queueing theory are reviewed in this chapter.

Chapter 3 compares two commonly used buffer sharing models for an SDN switch. These are the shared buffer model and priority queueing buffer model. The comparison investigates trade-offs between these two buffer sharing mechanisms.

Chapter 4 investigates the effect of internal buffering in an SDN switch. The queueing model for an SDN switch with internal buffer is developed and compared against the queueing model for an SDN switch without the internal buffer. The benefits and trade-offs of using the internal buffer in an SDN switch are identified.

Chapter 5 investigates the effect of a hardware switching in an SDN switch. The queueing models for a hardware-based SDN switch with and without the internal buffer are developed. These models are compared against a software-based SDN switch to develop guidelines for the selection of a software data plane or a hardware data plane.

Chapter 6 concludes this thesis with a summary of findings and suggestions for future work based on the outcomes.

Chapter 2

Background and Related Work

This thesis investigates the packet processing aspects of the switch in the Software-Defined Networking (SDN) to improve the overall performance of an SDN. This chapter starts with the brief overview of an SDN to provide information about its concept, architecture, and one of its popular protocol – OpenFlow. It is then followed by an explanation of an SDN switch's packet processing aspects that need to be analysed. This is followed by an explanation of existing approaches for performance analysis of an SDN. The pros and cons of existing approaches are then described. The reasons for selection of analytical tools, queueing theory, and Quasi-Birth-Death process are briefly discussed in existing approaches for performance analysis.

Next, previous works that are related to queueing models for SDN switches are reviewed in detail. Finally, this chapter is concluded with the summary of background and related works.

2.1 SDN: Overview

In a networked system, switches/routers have two basic components: the control plane and the data plane. The control plane performs control logic such as routing protocols, middlebox configuration while the data plane forwards the traffic based on control logic. The control plane and data plane are coupled together, thus enabling each switch/router to participate in route making decisions and data forwarding.

However, this tight coupling is not flexible when the network size increases. The control logic coupled with the hardware and the use of low-level programming languages makes the system rigid and complex. Therefore it copes poorly with issues of scalability, reliability and security which are more pronounced as the network size increases. These issues in the traditional network hinder the performance of the system as the network traffic is increasing day by day.

In order to cope with these issues in the traditional network, a new network paradigm called as "Software-Defined Networking" (SDN) emerged. SDN has two defining characteristics: (a) it separates the control plane from the data plane, and (b) it logically centralizes the control plane to direct multiple data plane elements through a software program [22].

Figure 2.1 shows the comparison between traditional networking and SDN architectures. In an SDN architecture, the control plane is decoupled from the data plane whereas the control plane and data plane are coupled together in the traditional networking architecture [3].

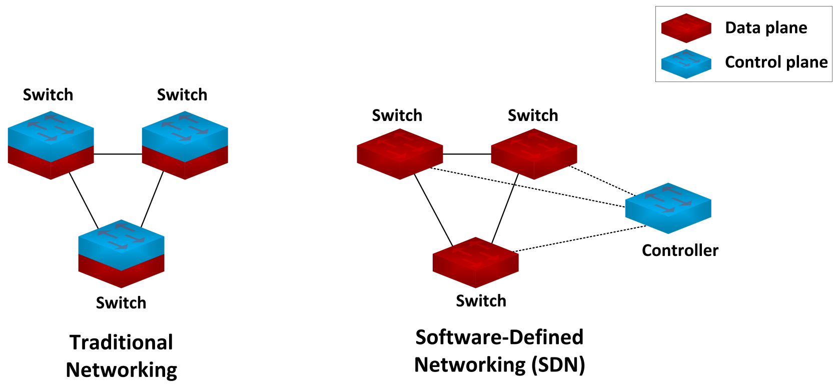


Figure 2.1: Comparison between traditional networking architecture and SDN.

The decoupling of control plane from data plane simplifies the function of a switch/router to that of forwarding traffic. The decoupled control is logically centralized and is named as the controller. Therefore, an SDN is the concept in which a network can be controlled and managed through software, for this reason it is termed as Software-Defined Networking. A switch/router can be configured remotely and network resources can be virtualized through the software.

A typical SDN consists of three components: a network element, an SDN Controller, and an SDN Application as shown in Fig. 2.2 [23].

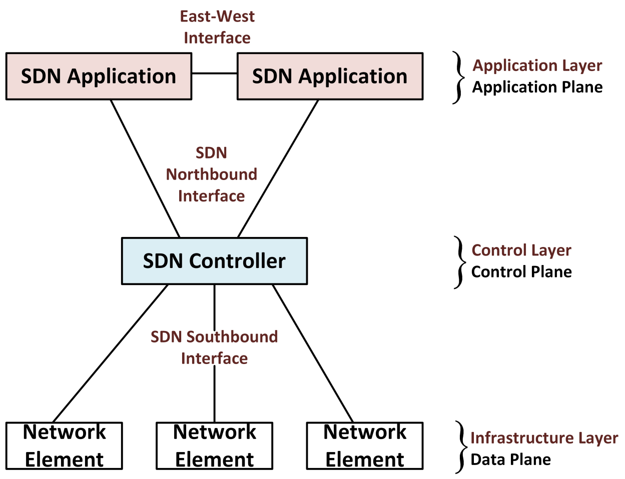


Figure 2.2: The components of a typical SDN.

- Network element: This is a forwarding device (switch or router) with a data plane only. A network element allows the controller to manage its functions via a southbound interface.
- SDN controller: This is a software entity that has network wide abstract and controls the network's basic functions like routing policy, middlebox configuration, etc.
- SDN application: This is the software program that directs the controller to perform the network's basic functions via a northbound

interface [24].

There are three basic types of interfaces in SDN architecture: Southbound interface, Northbound interface, and East-West interface.

- Southbound Interface: This is the interface between the network element, that is the SDN switch/router, and the controller. The SDN data plane can be programmed with this interface. Some protocols for the southbound interface are OpenFlow [25], ForCES [26], Border Gateway Protocol (BGP) [27], Network Configuration Protocol (NETCONF) [28], and Location Identifier Separation Protocol (LISP) [29]. OpenFlow, being a popular and widely used standard [3] is used for analysis in this thesis.
- Northbound Interface: This is the interface between the SDN controller and SDN application. With this interface, the SDN controller is directed to perform specific functions such as routing, firewall, and load balancer [30]. Some examples of the northbound interface are Floodlight Representational State Transfer (REST) API [31], OpenStack REST API [32], and Vyatta Remote Access API [33].
- East-West Interface: This is the interface between SDN controllers. With this interface, SDN controllers interact with each other. Currently, there is no standard for the east-west interface but efforts have been made to standardize it with gateway protocols like BGP (Border Gateway Protocol) [27], SIP (Session Initiation Protocol) [34], SDNi [35], and East-West Bridge [36].

OpenFlow is the southbound interface standard protocol proposed by the Open Network Foundation (ONF) [37]. The ONF is a user-driven organization for developing and standardizing SDN. It defines communication between the control plane and data plane in SDN by allowing programmability through the flow table contained in a switch/router. In the OpenFlow architecture [25], the OpenFlow switch maintains one or more flow tables. Flow tables are linked together to form a pipeline, where each flow table has flow table entries (FTE) which contain a set of match fields. These match fields are matched against the incoming packet and a specified action is applied. Every flow table has a table-miss entry which is

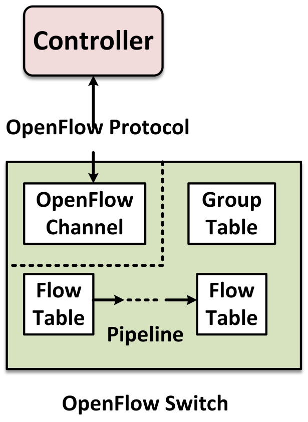


Figure 2.3: OpenFlow-based SDN architecture with one controller and one switch.

used if there is no matching flow table entry. This table-miss entry specifies what to do with an unmatched packet either by sending it to the controller, dropping it or directing it to next table. Figure 2.3 shows the OpenFlow-based SDN architecture with one controller and one switch.

2.2 Packet processing in an SDN switch

A computer communication network comprises of switches that forward packets from source to destination. The performance of any computer network largely depends on packet processing in a switch. For instance, faster processing switches can forward packets faster which reduces the overall delay in the network. Similarly, switches with sufficient memory can process a larger number of packets via internal buffering, thereby preventing loss of packets in the network. Therefore, it is important to analyse packet processing aspects of a switch to obtain desirable QoS (Quality of Service) and improve the overall performance of a network.

SDN was introduced to simplify forwarding of packets. Switches in an SDN can be programmed through software that makes a computer network agile and flexible. SDN switches can be used in time-critical systems like industrial automation [19], interactive video [18], and online surgery [20] that require extremely low packet losses (i.e. below 10^{-4}) [38]. However, design issues such as provisioning of switch capacity and buffer sharing mechanisms significantly impact the overall performance of an SDN. To address switch design issues and provide guidelines to network operators, various aspects of packet processing in an SDN switch have been analysed. These aspects include buffer sharing mechanisms, packet encapsulation methods, and SDN switch types which are discussed in the following subsections.

2.2.1 Buffer sharing mechanism

Decoupling the control and data plane presents a challenging opportunity for research in buffering requirements due to the different time scales of packet processing and traffic volume in the data plane and control plane. Packet processing in an SDN switch is categorized into: (i) fast path - referring to data plane processing (e.g. checksum calculation, TTL (time to live) decrement), and (ii) slow path - referring to the control plane processing, switch management functions, and exception processing (e.g. IP lookup). Packets traversing the fast path expect low latency, typically in orders of hundreds of nanosceconds while packets traversing the slow path experience tens of miliseconds delay.

Packet delays in a switch are determined by the capacity and service rate of the output buffers because output buffers in a switch are shared by all ports. Shared buffer switches are the most cost effective because multiple input ports share a single output buffer and reduces the cost and complexity of buffers to individual output ports in the switch. Alternatively, two separate buffers each serving the fast path and slow path can be used to buffer packets.

2.2.2 Packet encapsulation at an SDN switch

A packet encapsulation in the OpenFlow-based SDN is the encapsulation of an entire data packet or part of a data packet in the switch with an asynchronous message designated as "packet-in". A packet-in message is sent to the controller without a request from the controller when there is no matching flow table entries for an external data packet arriving in the switch [13]. Based on the configuration of a packet-in message, a packet encapsulation at an SDN switch can be categorized into three types:

- Full encapsulation of a packet (E): If an SDN switch does not support internal buffering due to insufficient memory, then full packet is encapsulated with a packet-in message. The full encapsulation method is the default packet processing used at an SDN switch.
- Internal buffering of a packet (I): If an SDN switch has sufficient memory to buffer packets, then part of each buffered packet (i.e. packet header that contains routing information which is used by the controller to make forwarding decisions) along with an identifier (i.e. buffer ID) is encapsulated with a packet-in message. In this case, a packet to be internally buffered is temporarily queued at the switch processor.
- Partial encapsulation of a packet (E-I): If an SDN switch with limited memory supports internal buffering, then packets will be queued at the switch processor until it has free available memory. When it runs out of memory to buffer packets, then a packet will be fully encapsulated with a packet-in message. Hence, this case is termed as the partial encapsulation of a packet.

In this thesis, "E" and "I" packet encapsulation methods are studied.

2.2.3 SDN switch types

SDN switches are categorized into Software-based Switch and Hardware or Physical Switch [4]. An SDN software switch maintains the flow table in SDRAM (synchronous dynamic random access memory) where the incoming packet is matched against the FTE using a CPU (central processing unit). If there is no matching FTE, a packet is forwarded to the controller which will feedback forwarding information to the switch and update the software flow table. The packet processing logic in a software switch is implemented in software [4] usually with the help of optimized software libraries. Open vSwitch (OVS) [39], Pantou/OpenWRT [40], ofsoftswitch13 [41], and Indigo [42] running on commodity hardware (e.g. desktops with several network interface cards) are a few examples of SDN software switches.

In an SDN hardware switch, the packet processing function is embedded in the specialized hardware. This specialized hardware includes layer two forwarding tables implemented using content-addressable memories (CAMs), layer three forwarding tables using ternary content-addressable memories (TCAMs) [4] and application-specific integrated circuits (ASICs). In a hardware switch, the FTEs are stored in CAMs and TCAMs of specialized hardware and packets are processed by the ASICs. Hardware switches are also equipped with SDRAM and CPU allowing a hardware switch to maintain flow tables in both TCAM and SDRAM [14]. Switches such as the Mellanox SN2000 series, NoviFlow NoviSwitch class of switches, HP ProCurve J9451A, Fulcrum Monaco Reference, Quanta LB4G, and Juniper Juno MX-Series are classified as hardware switches [43, 3].

In the following section, different approaches that are available for the performance analysis of an SDN network are discussed. The benefits and limitations of these approaches are briefly discussed followed by the reasons for the selection of analytical modelling.

2.3 Approaches for performance analysis

Computer communication networks are becoming complex and more advanced with the digital revolution. The technological advancement has changed different aspects of human life from education, research, development, and business [17]. Computer network designers are constantly working to improve the communication systems with the help of modelling and performance analysis tools. The performance analysis tool gives freedom and flexibility to network designers to study and predict a system before it's implementation.

As SDN is a new networking architecture, various design issues like traffic types (control and data traffic), switch buffer capacities, and buffer sharing mechanisms that can impact the overall performance of a network should be identified and analysed. For this purpose, we have to do performance analysis to identify the potential bottlenecks that can hinder the overall performance of an SDN network. The three commonly used performance measures for an SDN network are:

- Delay: This is the time that it takes to transmit a packet from a sender to a receiver.
- Throughput: This is the rate of a total number of packets transmitted to a receiver by a sender.
- Packet loss rate: This is the number of packets being blocked or dropped out of the total packets transmitted by a sender. It is expressed in a percentage.

In computer communication networks, these performance measures have a huge impact on the overall user experience. Delay and throughput determine the speed of the network while the packet loss rate is a negative indicator that causes a breakup in the communication.

The performance analysis of an SDN network has been done in three different ways: Experimental testbeds, Software-based tools, and Mathematical models [15]. From the survey shown in Fig. 2.4 that there are not many mathematical models and experimental testbeds available for performance analysis of an SDN-based network. A hundred papers were surveyed which focused on SDN using the following keywords in Google Scholar, ACM Digital Library, IEEE Xplore, and SpringerLink: experimental testbeds, simulation, emulation, measurement tools, analytical models, performance analysis.

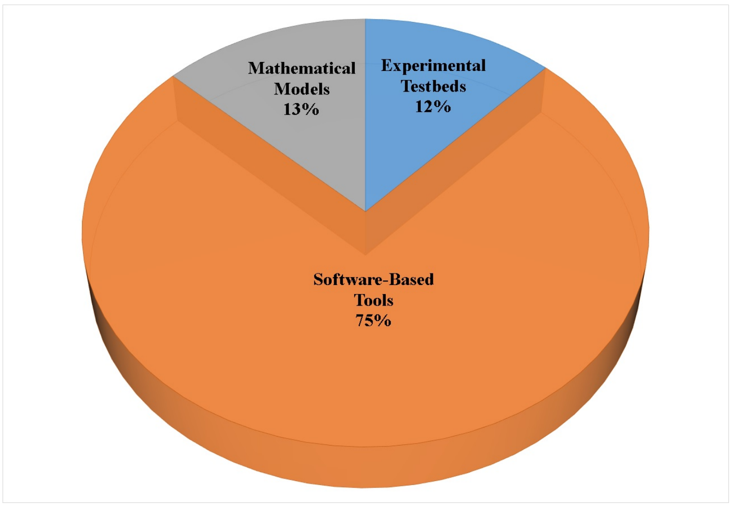


Figure 2.4: Approaches for performance analysis in SDN

2.3.1 Experimental testbeds

Experimental testbeds are real-time hardware testbeds which allow the researchers to validate and test their algorithms and mechanisms. Some of the examples of experimental testbeds used for an SDN are GENI (Global Environment for Networking Innovations) in the USA [44]; AKARI [45], JGN-X (Japan Gigabit Network-X) [46], and RISE [47] in Japan; FEDERICA (Federated E-Infrastructure Dedicated to European Researchers Innovating in Computing Network Architectures) [48], OFELIA (OpenFlow in Europe: Linking Infrastructure and Applications) [49], FIBRE (Future Internet Research and Experimentation) [46], and OpenLab [50] in Europe;

OF@TEIN [51] in South Korea and JOLNET [50] in Italy.

Experimental testbeds provide a flexible environment for testing, measuring, and validating new networking technologies and applications with real traffic. They can also use artificial traffic with packet manipulation tools like Scapy [52]. Experimental testbeds use high-performance real devices which make them more convincing than emulators and simulators [50]. However, experimental testbeds are expensive to deploy and have scalability problems due to hardware limitations [2, 50, 53, 54].

2.3.2 Software-based tools

Software-based tools are used to debug, verify and simulate or emulate SDN applications and APIs [55]. Debugging and testing tools ensure the quality of an SDN software by getting rid of bugs and errors [56]. Some of the available debugging and testing tools are FlowChecker, ndb, Veriflow, OFRewind, OFf, OFTest, NICE, NetSight, PathletTracer, and SDN traceroute [2, 56]. Similarly, verification tools are used to verify and analyse the characteristics of SDN switches and controllers. Some of the available verification tools for an SDN are Cbench, OFCBenchmark, OFLOPS, OFLOPS-Turbo, OpenSketch, SDLoad [2].

A simulator is software that sets up a necessary environment to simulate the hardware's behaviour, while an emulator emulates every aspect of hardware's behaviour to make it an exact replica of actual hardware. Some examples of the simulator used in an SDN are NS-3, FS-SDN, MaxiNet, STS,VND-SDN, EstiNet 8.0 OpenFlow netowrk simulator, and OMNet++ [2]. Mininet, Mininet CE, Mininet-HiFi and DOT[2] are examples of emulator. Simulators and emulators are low-cost and flexible compared to experimental testbeds. However, simulators use artificial traffic while emulators use real traffic [2, 3, 55, 56].

2.3.3 Mathematical models

For the networks to be designed or operational networks that need to be optimized, a mathematical-based performance analysis is the cost-effective solution to predict performance measures [16]. The mathematical model represents the system in the form of mathematical equations, theorems and proofs. The model can be used to support experimental or simulated results. It also has the advantage of being cheaper and faster than the experimental framework or simulator. However, there are very few models available for mathematical analysis of an SDN-based network. The two basic mathematical methodologies used to model an SDN-based network are queueing theory and network calculus. The queueing theory shows the performance of a system in an average quantities at equilibrium state, while the Network Calculus shows the performance of a system in a probabilistic bound curve with the worst case scenario.

2.3.3.1 Network Calculus

An SDN-based network has two level of services: flow-level and packet-level. A flow-level service is for the SDN-controller while a packet-level service is for an SDN switch [57]. The network calculus is used in an SDN for performance analysis of a flow-level arrival process which may consist of multiple or many packet-level arrival processes.

Network Calculus uses min-plus algebra and max-plus algebra to transform complex network systems into an analytical system that can be tractable [58]. Network Calculus is used for modelling a flow as a cumulative arrival process A. $A(t)$ represents the total number of packet arrivals in the interval $[0, t)$ [59].

The cumulative arrival process A at time "t" from any time "s" is characterized by the average sustainable arrival rate (ρ) and the burstiness (σ) such that $A \sim (\sigma, \rho)$:

$$A(t) - A(s) \leq \sigma + \rho(t - s), \quad 0 \leq s \leq t. \tag{2.1}$$

If constrained flows are merged, the resulting process is also constrained according to the multiplexing rule:

$$A_i \sim (\sigma_i, \rho_i) \rightarrow \sum A_i \sim (\sum \sigma_i, \sum \rho_i). \tag{2.2}$$

At time τ, the stopped sequence (A^τ) is defined for increasing sequence A.

$$A^\tau(t) = \begin{cases} A(t), & \text{if } t \leq \tau, \\ A(\tau), & \text{otherwise.} \end{cases} \tag{2.3}$$

If there are no packet arrivals after time τ, the stopped sequence A^τ is equal to $(\sigma(t), \rho)$ where

$$\sigma(t) = \max_{0 \leq t \leq \tau} \max_{0 \leq s \leq t} [A(t) - A(s) - \rho(t-s)]. \tag{2.4}$$

The summary of analytical models that use Network Calculus to model an SDN network is shown in Table 2.1. The analytical model presented in [60] has used network calculus to characterize the behaviour of the control interface with a single controller and switch. However, this model does not consider the interaction between the switch and controller. The interaction between switch and controller was considered in [57] with feedback model that computes worst-case bounds on performance metrics. The work in [57] was extended in [61] for scalable SDNs with multiple controllers. Similarly, [62] proposed the closed form expression to compute the worst-case bounds for the distributed control plane. The work in [63] proposed a hybrid scheduling model that combines PGPS (Packet Generalized Processor Sharing) and preemptive priority scheduling algorithms for multimedia flows in switches.

In the above-mentioned analytical models, a flow-level arrival in the controller and a packet-level arrival in the switch has been considered. However, for an SDN network where we want to study and analyse an SDN switch, queueing theory is a more useful mathematical tool to perform a packet-level analysis.

Table 2.1: Related works on SDN modelling using Network Calculus.

Model	**Contribution**
Bozakov, 2013 [60]	Model to characterize the service of the switch's control interface.
Azodolmolky, 2013 [57]	Feedback model to characterize interaction between the switch and controller.
Azodolmolky, 2013 [61]	Worst delay bound case for Scalable SDNs.
Koohanestani, 2017 [62]	Delay bound based on the similarities between caches and flow tables in switches.
Huang, 2017 [63]	Hybrid scheduling model for multimedia flows in switches.

2.3.3.2 Queueing theory

Queueing theory is the classical mathematical tool to study queues. It has been used for performance analysis and modelling of computer networks for many years where a computer network is represented as a network of queues [64]. It is used to analyse a network's delay and throughput at the steady state. In queueing theory, Kendall notation i.e $A/S/c/K/D$ [65] is used to describe and classify a queue where

- A denotes inter-arrival time distribution,
- S denotes service time distribution,
- c denotes the total number of servers in the system,
- K denotes the maximum number of packets in the system, and
- D denotes the queue discipline.

The distribution of A and S may be Exponential Distribution (M), Deterministic Distribution (D), Erlang Distribution (E_k), General Distribution (G), or Phase-type Distribution (PH), to name a few. Similarly, the common queue disciplines are FIFO (First In First Out), LIFO (Last In First

Out), SIRO (Service In Random Order), PNPN (Priority Service) and PS (Processor Sharing).

$M/M/1$ is the simplest queueing model where the system has one server with the arrival of Poisson distribution, service of exponential time distribution, infinite capacity and FIFO as default queue principle. The basic performance metrics determined using queueing theory in a computer network are delay and throughput.

The summary of analytical models that uses Queueing theory to analyse an SDN network is shown in Table 2.2. The model in [66] is the first analytical model for an SDN network with single switch and controller where switch (as M/M/1 queue) and controller (as M/M/1/K queue) are assumed to be operating independently. This assumption of independent operation was removed in [67] with the switch and controller collectively modelled as a Jackson's network. The work in [67] was extended in [68] with multiple switches. The model in [69] have modelled SDN-based cloud computing as a two-stage tandem network where the switch is modelled as M/M/1 queue with no distinction between the control and data traffic.

The models in [12] and [11] are the first to use priority buffer sharing models in the switch. The model in [12] uses a preemptive priority buffer while the model in [11] uses non-preemptive priority buffer in the switch. In both of these models, priority is given to the control packet over the data packet. However, the use of non-preemptive priority shows the accurate representation of an SDN behaviour where a packet in the processor is serviced without interruption. The work in [12] was extended in [70] with arrival assumed as an MMPP under bursty multimedia traffic scenario. Similarly, [71] have followed the work in [11] and have compared shared buffer with non-preemptive priority buffer sharing model in the switch. In [71], with the help of buffer dimensioning, they have shown that the use of priority buffer requires lower time to install FTE compared to a shared buffer model.

The model presented in [72] has assumed the switch service time distribution as a two-phase hyperexponential distribution and modelled switch as $M/H_2/1$ queue which has been studied earlier in [73]. Similarly, the model presented in [74] also have assumed switch as $M/H_2/1$ queue and proposed a network visualization and performance evaluation model based on [73] and [75]. In [72] and [74], two different service distributions for packets arriving at the switch are assumed. One of the service distributions is used to forward packets with matching flow updates, while the other one is used to send packets without matching flow updates to the controller. The use of two different service distributions paves the way for modelling hardware switches, one for the CPU and the other one for the TCAM.

The model presented in [76] have assumed the switch as M/G/1 model with log-normal mixture model as the service distribution. The work in [76] further demonstrates M/M/1 as a poor fit for the switch through experiments. The model presented in [77] is the first to model SDN hardware switch but does not consider a hardware data plane. In [77], the switch is assumed as M/Geo/1 model where inter-arrival is exponentially distributed and the service time is geometrically distributed.

The model presented in [78] models TCP connections over an SDN where both the switch and controller are modelled as MMPP/M/1. In [78], four-dimensional states are used to evaluate the performance of the switch and controller jointly under steady states, hence the model is named as the 4D state model.

The models presented in [79] are the first to model an SDN with NFV (Network function virtualization). In this work, two models: NFV under the controller, and NFV aside the controller are analysed. The comparative analysis in [79] shows that NFV aside the controller is a better architecture for integrating an SDN with NFV. The model presented in [80] is the exception that does not consider the switch and focused only on interactions between controllers.

Table 2.2: Related works on SDN modelling using Queueing Theory.

Model	Switch	Contribution
Jarschel, 2011 [66]	M/M/1	First to model an SDN where the switch and controller operate independently.
Mahmood, 2014 [67]	M/M/1	The switch and controller are collectively modelled as Jackson's network.
Yen, 2014 [69]	M/M/1	SDN-based cloud computing as a two-stage tandem network.
Mahmood, 2015 [68]	M/M/1	Extension of [67] with multiple switches.
Wang, 2015 [80]	–	Only focuses on controllers, root and local controllers.
Miao, 2015 [12]	M/M/1	Compare shared and preemptive priority buffer sharing models in the switch.
Shang, 2016 [72]	M/H_2/1	Two different service distributions for packets arriving at the switch.
Xiong, 2016 [73]	M^X/M/1	Investigated the arrival flow request messages at the controller.
Miao, 2016 [70]	MMAP	Extension of [12] with arrival as MMAP under bursty multimedia traffic scenario.
Sood, 2016 [77]	M/Geo/1	Among the first to model an SDN hardware switch.
Goto, 2016 [11]	GI/M/1/K	Non-preemptive priority buffer model in the switch.
Javed, 2017 [76]	M/G/1	Switch as M/G/1 with log-normal mixture model as the service distribution.
Muhizi, 2017 [74]	M/H_2/1	Network visualization and performance evaluation model.
Singh, 2017 [71]	GI/M/1/K	Compare shared and non-preemptive priority buffer sharing models in the switch with buffer dimensioning.
Lai, 2017 [78]	MMPP/M/1	TCP connection over an SDN.
Fahmin, 2018 [79]	M/M/1	An SDN with NFV.

In this thesis, we use queueing theory over Network Calculus to analyse an SDN switch. This is because the Network Calculus analyses worst case conditions of network which may result in over-provisioning of resources in real world production [62]. In queueing theory, the Quasi-Birth-Death (QBD) process has been widely used to model a computer network in greater detail [16]. Some examples that use QBD for detailed study of a computer network are scheduling of resource reservation [81], reliability of a computer system [82], telecommunication model with impatient customers [83], and modelling of P2P file sharing systems [84]. To study an SDN switch in greater detail, the modelling approach in this thesis is based on QBD processes. Hence, the following section is devoted to describe the notation and concepts behind QBD processes.

2.3.4 Quasi-Birth-Death process

A continuous-time Markov chain with multidimensional state spaces that can be partitioned into disjoint levels is a QBD process [85]. A continuous-time QBD process is a two-dimensional Markov chain represented as

$$\{(X_t, Y_t), t \geq 0\} \tag{2.5}$$

with the state space $\mathbb{S} = \{(i, j) \in \{0, 1, ..., K\} \times \{0, 1, ..., L\}\}$ where i and j denote the level and phase variables of the process, respectively [86]. Similarly, K and L determine the queue capacities of level and phase variables respectively which can be finite or infinite. In an SDN network, the controller may be represented by a level variable and the switch by a phase variable.

In queueing network, QBD processes can be multi-dimensional with one level variable and multi-dimension phase variables based on the number of the nodes or queues in the network. For N queues, the state of the network can be represented by the vector $n = (n_1, n_2, ..., n_N)$ where n_l is the number of packets in queue l. If queue 1 is the queue of interest for analysis, then packets at queue 1 (i.e. n_1) are represented by the level variable and packets at queues other than queue 1 are represented by phase variables as the vector $r = (n_2, n_3, ..., n_N)$ [87].

In QBD processes, the transitions between the state are limited within the level or between two adjacent levels. If the transitions of QBD process are independent of level, then such type of QBD process is homogenous or level-independent. Similarly, if the transitions are dependent of level, then QBD process is nonhomogenous or level-dependent [88].

2.3.4.1 Homogenous QBD process

Using standard QBD notation [89], the transition matrix for a homogenous or level-independent QBD process is given by an infinitesimal generator matrix (Q) with a repetitive tri-diagonal block structure as shown below:

$$Q = \begin{pmatrix} B_1 & A_0 & & & & \\ A_2 & A_1 & A_0 & & & \\ & A_2 & A_1 & A_0 & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_2 & A_1 & A_0 \\ & & & & A_2 & A_1 \end{pmatrix}, \tag{2.6}$$

where $A_0, A_1, A_2,$ and B_1 are standard notations of block matrices used to represent phase variable distributions in homogeneous QBD processes. The matrices B_1 and A_1 represent the phase distributions for boundary condition (i.e. $i = 0$) and non-boundary condition (i.e. $i \neq 0$) respectively when level variable remain unchanged (i.e. $i \rightarrow i$). Similarly, A_0 and A_2 represent the phase distributions when level variable increases (i.e. $i \rightarrow i + 1$) or decreases (i.e. $i \rightarrow i - 1$ for $i > 0$) by 1, respectively. In an SDN network, A_1 or B_1, A_0, and A_2 represent homogeneous state distribution of the switch when the number of packets in the controller remains unchanged, increases by 1, and decreases by 1, respectively.

With the help of Matrix Geomteric/Analytic Method [87], the stationary probability distribution π of homogeneous QBD process can be computed that satisfies the system of equations $\pi Q = 0$ and $\pi e = 1$, where "e" is the column vector of ones. The stationary probabilities are defined as:

$$\pi_{ij} := \lim_{t \to \infty} \mathbb{P}(X_t = i, Y_t = j), \tag{2.7}$$

where $\pi_i = (\pi_{i0}, \pi_{i1}, ..., \pi_{iL})$, for $i = 0, 1, ..., K$ and $\pi = (\pi_0, \pi_1, \pi_2, ...)$. The stationary probabilities can be used to compute network performance such as throughput, delay, and packet loss rate.

The homogeneous QBD process is positive recurrent and markov chain is ergodic if

$$\pi_A A_0 e < \pi_A A_2 e, \tag{2.8}$$

where π_A is the stationary probability vector of the matrix A which equal the sum of matrices A_0, A_1 and A_2. For ergodic markov chain, the system of equation $\pi Q = 0$ is equivalent to second-order difference equations

$$\pi_0 B_1 + \pi_1 A_2 = 0, \tag{2.9}$$

$$\pi_{i-1} A_0 + \pi_i A_1 + \pi_{i+1} A_2 = 0, \quad i \geq 1, \tag{2.10}$$

with $\pi_i = \pi_{i-1} R$ for $i \geq 2$, where R is the rate matrix of QBD process with dimensions $(L+1) \times (L+1)$ and is non-negative solution to the equation

$$A_0 + R A_1 + R^2 A_2 = 0. \tag{2.11}$$

The matrix R can be computed using various iterative algorithms like Successive Substitution (SS) [89], Logarithmic Reduction (LR) [90], and Cyclic Reduction (CR) [91]. However, the LR method has been widely used in performance analysis of communication systems due to a faster quadratic convergence rate [16, 92].

The LR method computes matrix G instead of directly computing the rate matrix R. The matrix G represents the hitting probability distribution on states in a given level and has an intuitive stochastic interpretation for ergodic QBD process [87]. Like matrix R, matrix G is also characterized as non-negative solution which is dual equation to Eq. 2.11,

$$A_2 + GA_1 + G^2A_0 = 0. \tag{2.12}$$

The computation of matrices G and R using the LR method is shown in Appendix A.

2.3.4.2 Nonhomogenous QBD process

The nonhomogeneous QBD process has been widely used to model telecommunication systems with dynamic stochastic arrivals and holding times [93]. It has level-dependent transition rates and is given by infinitesimal generator matrix Q using standard notation with a repetitive block structure [89],

$$Q = \begin{pmatrix} A_1^{(0)} & A_0^{(0)} & & & & \\ A_2^{(1)} & A_1^{(1)} & A_0^{(1)} & & & \\ & A_2^{(2)} & A_1^{(2)} & A_0^{(2)} & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_2^{(K-1)} & A_1^{(K-1)} & A_0^{(K-1)} \\ & & & & A_2^{(K)} & A_1^{(K)} \end{pmatrix}, \tag{2.13}$$

where $A_0^{(i)}$, $A_1^{(i)}$, and $A_2^{(i)}$ are non-negative sub-matrices for $i \geq 0$. The sub-matrices $A_0^{(i)}$, $A_1^{(i)}$, and $A_2^{(i)}$ represent the phase variable distributions when level variable increases by 1 (i.e. $i \to i+1$), remains unchanged (i.e. $i \to i$), and decreases by 1 (i.e. $i \to i-1$ for $i > 0$), respectively.

Similar to homogeneous QBD, the system of equations $\pi Q = 0$ for nonhomogeneous QBD process is equivalent to the second-order matrix difference equations

$$\pi_0 A_1^{(0)} + \pi_1 A_2^{(1)} = 0, \tag{2.14}$$

$$\pi_{i-1}A_0^{(i-1)} + \pi_i A_1^{(i)} + \pi_{i+1}A_2^{(i+1)} = 0, \text{ for } i = 1, 2, ..., K-1, \tag{2.15}$$

$$\pi_{K-1}A_0^{(K-1)} + \pi_K A_1^{(K)} = 0, \tag{2.16}$$

with first-order recurrence scheme i.e. $\pi_{i+1} = \pi_i R_i$ for $i = 0, 1, ..., K-1$. Substituting $\pi_{i+1} = \pi_i R_i$ in Eqs. 2.14, 2.15, and 2.16 yields

$$\pi_0(A_1^{(0)} + R_0 A_2^{(1)}) = 0, \tag{2.17}$$

$$\pi_{i-1}(A_0^{(i-1)} + R_{i-1} A_1^{(i)} + R_i A_2^{(i+1)}) = 0, \text{ for } i = 1, 2, ..., K-1, \tag{2.18}$$

$$\pi_{K-1}(A_0^{(K-1)} + R_{K-1} A_1^{(K)}) = 0, \tag{2.19}$$

Equations 2.17, 2.18, and 2.19 are the solution to stationary distribution probabilities for the nonhomogeneous QBD process which requires the rate matrix for the highest level (R_K).

There are various methods to compute R_K, out of which the Bright-and-Taylor method [94] and the Matrix Continued Fraction Algorithm (MCF) [95] are widely used. The Bright-and-Taylor method is among the first to compute rate matrices for nonhomogeneous QBD by approximating R_K. This approximation of R_K requires larger memory for computation and storage of iteration matrices to satisfy specified tolerance (say 10^{-10}). This may result in overflow or underflow errors in a memory-constrained network [83]. As an alternative, MCF method is faster and more efficient which assumes R_K as a zero matrix instead of approximating it. The algorithm for the MCF method is shown in Appendix B.

2.4 Related works on queueing models for an SDN Switch

This section is devoted to detail existing works that use queueing theory as analytical tool to model an SDN network. This section will start with a generic queueing model for an SDN network. Based on this generic queueing model, we will relate existing works into three different aspects of a performance analysis for an SDN switch as mentioned in the Section 2.2: (1) Buffer Sharing Models for *Buffer Sharing Mechanism*, (2) Internal Buffering for *Packet Encapsulation*, and (3) SDN switches for *SDN*

Switch Types.

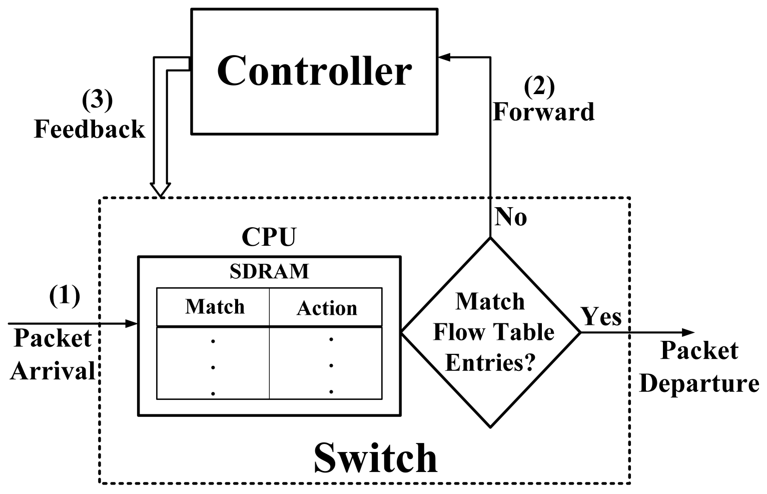


Figure 2.5: Generic model for an SDN network.

A generic block diagram of an SDN network where an external packet arrives at the switch and the switch is connected to a controller is shown in Fig. 2.5. There are three important phases an SDN model with a switch must capture. Phase (1), the first packet of a flow arrives at the switch and there is no matching FTE for the packet in SDRAM. Phase (2), the packet without a matching FTE is forwarded to the controller **or** a packet with the matching FTE is serviced by the switch and forwarded to the destination. All packet processing and forwarding in the switch is executed on CPU and SDRAM. Finally, Phase (3), the controller feeds the forwarding information back to the switch and updates the flow table in the switch.

2.4.1 Buffer sharing models

Bufferring in a switch is primarily to absorb temporary traffic fluctuations. The output buffers of the switch block in Fig. 2.5 is modelled either as: (i) a single shared queue or (ii) a two-priority queue (fast path vs. slow path as discussed in Section 2.2.1). In the shared queue model, packets passing through the fast path and slow path share a single queue with first in first out (FIFO) service discipline. While in the two-priority queue, the fast path and slow path packets are queued separately and each queue is served in a FIFO manner without preemption. In the two-priority queue model, packets in the slow path are always served with higher priority over fast path packets but the server does not preempt the service of fast path packets once it has commenced, this is referred to as service without preemption.

The analytical model in [66] models the switch block as a shared M/M/1 queue and the controller block represented by M/M/1/K queueing system, respectively. In this model, the controller and switch queues are considered to be operating independently and a fraction of incoming external traffic to the switch is forwarded to the controller. This independence assumption together with the infinite buffers preserves the Markovian property of the individual queues, reduces the complexity in the resulting queueing network and yields a model that is amenable to product-form analysis. Product-form analysis decomposes the stationary probability distribution of the queueing network into the product of marginal probability distribution of each queue (Burke's theorem [96, 97]), thus greatly simplifying analysis.

While the shared queue model simplifies analyses, it does not reflect the realities of the different packet processing time scales of the fast path vs. slow path. There is an implicit assumption in the shared queue model that the traffic on the slow path forwarded by the controller *back* to the switch is treated identically to new packets entering the switch. This is a strong assumption because the control vs. data plane distinction is not taken into consideration.

The model presented in [67] considers switch and controller collectively as Jackson's network mimicking the model in [66]. The model in [67] differs from [66] in that it does not assume that the controller and switch are independent. The traffic forwarded by the controller to the switch mixes with the external packet arrivals to the switch and this mixing better reflects the operational realities of an SDN switch.

The model presented in [69] considers an SDN-based cloud computing as a two-stage tandem network. In this work, the switch is assumed to be a M/M/1 model with simple approximate analysis and does not distinguish the control and data traffic.

The model presented in [76] demonstrates M/M/1 as a poor fit for the switch through experiments and have assumed switch to be a M/G/1 model. Similarly, the model presented in [78] models TCP connections over an SDN with both the switch and controller as MMPP/1/1. However, the analysis in [78] is an approximation with no clear distinction between control packets and data packets.

The model presented in [12] uses preemptive priority queues to represent a switch with infinite capacity. This work also compares a priority queueing buffer and a shared buffer using a simpler analysis concluding priority queueing to be a better buffer sharing mechanism. However, the work in [12] does not analyse the buffer dimensioning which can be a tradeoff between the selection of priority queueing buffer and shared buffer in an SDN switch. The work in [12] is extended into [70] which assumes arrival to be a MMPP under a bursty multimedia traffic scenario, and have also assumed infinite capacity for the high priority queue and finite capacity for the low priority queue.

The models presented in [11, 98] were the first to use priority queues and finite capacity respectively to represent switch. The priority queues in these models are non-preemptive whereby the lower priority queue is serviced when there are no packets in the higher priority queue. This queue structure more accurately reflects an OpenFlow switch. However, with the use of finite capacity queue model for the switch, the queueing model does not admit a product form queueing solution which is significantly more difficult to analyse. The tradeoff is clear: increased model complexity for more accurate representation of the SDN behavior.

Table 2.3: Summary of queueing models for buffer sharing mechanisms.

Model	**Buffering**		**Switch**	**Dimen-**	**Analysis**	
	SSB	PQ	**model**	**sioning**	EXA.	APP.
Jarschel [66]	✓		M/M/1			✓
Mahmood [67]	✓		M/M/1			✓
Yen [69]	✓		M/M/1			✓
Miao [12]	✓	✓	M/M/1			✓
Miao [70]		✓	MMAP			✓
Goto [11]		✓	GI/M/1/K		✓	
Javed [76]	✓		M/G/1			✓
Lai [78]	✓		MMPP/M/1			✓

SSB denotes single shared buffer, PQ denotes priority queue, EXA. denotes exact analysis and APP. denotes approximate analysis.

A summary of existing queueing models for buffer sharing mechanisms is shown in Table 2.3. In Table 2.3, the first column denotes existing queueing models for buffer sharing in SDN switches, the second column indicates buffer sharing mechanisms, the third column denotes the queueing model in Kendall's notation, the fourth column indicates model is used for buffer dimensioning in a switch and the final column denotes the type of analysis.

From Table 2.3, it is clear that most of the analysis is an approximation with the assumption of infinite buffer capacity in the switch. Similarly, only a few have used the priority queueing buffer model to accurately represent an SDN behaviour. However, none of the surveyed analytical models do buffer dimensioning which can be used to identify trade-offs between shared and priority buffer models.

2.4.2 Internal buffering

In the previous section, the related works for buffer sharing models have used the full encapsulation of data packets i.e. “E” (as discussed in Section 2.2.2) at an SDN switch. This is because “E” is the default packet encapsulation method in the OpenFlow protocol. However, OpenFlow specifications have provisions for switches to support internal buffering of data packets [13]. In this section, the related works on a packet encapsulation method that supports internal buffering in an SDN switch i.e. “I” (as discussed in Section 2.2.2) are discussed.

As discussed in Section 2.2.2, asynchronous messages are sent either with the arriving packet or only with a fraction of the packet header based on the availability of memory in the switches for internal buffering. In a packet, only the header contains routing information which is used by the controller to make forwarding decisions. If the switches have sufficient memory to buffer packets, then the packet header along with a buffer ID is sent with the asynchronous message. Similarly, some switches do not support internal buffering and require full packet (not just the header) to be sent with the asynchronous message.

While internal buffering has been well studied in a traditional switch, the buffering of asynchronous messages over a separated control and data plane remains unexplored. The separation of the data plane and control plane in an SDN brings a different set of challenges for switch designers working with SDN switches. For example the control decisions from the controller may take up to 1 millisecond to reach the switch.

Figure 2.6 shows the simple example of an OpenFlow switch that supports internal buffering. In Fig. 2.6, OpenFlow switch consists of a flow table for storing flow information and the CPU for processing the packets in a flow. The “packet-in” message is an asynchronous message sent by an OpenFlow switch to the controller while the “packet-out” message is controller-to-switch message [13]. The packet-in message generally consists of three important fields: header, buffer ID, entire packet or part of a

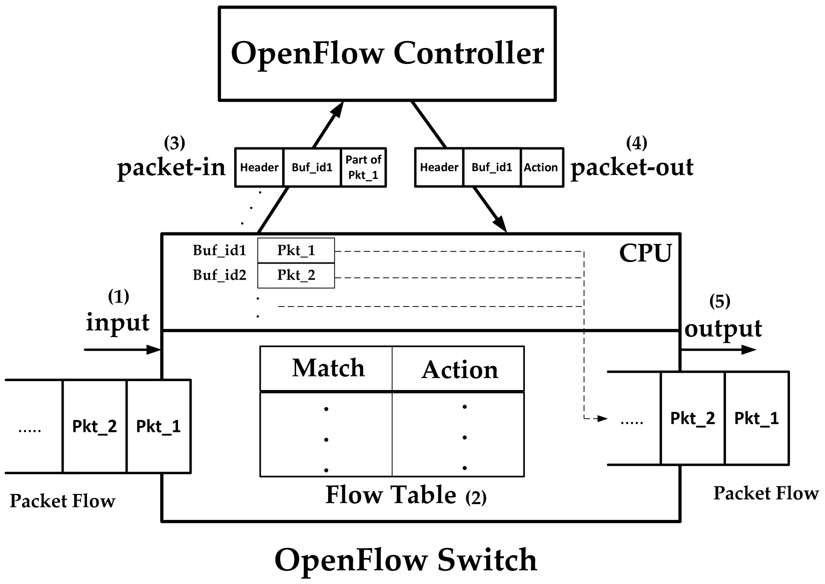


Figure 2.6: OpenFlow switch with support for internal buffering [1].

packet [1]. The header is OpenFlow header information, buffer ID is the ID given to the temporarily buffered packet in the CPU. The buffer ID field is only used when packet-in events are configured for the internal buffering. To simplify notation in Fig. 2.6, a packet with identifier is represented as "Pkt_#" and buffer ID as "Buf_id#". It is also assumed that packet_out message carries all control information including flowmod messages.

The packet processing for the internal buffering in an OpenFlow switch can be explained in five steps as shown in Fig. 2.6. First, an external data packet arrives at the switch. Second, the forwarding information in an external packet is matched against the flow table. Third, the data packet without matching FTE is temporarily buffered in the CPU and the packet-in message is sent to the controller **or** the data packet with matching FTE is outputted to the destination. Fourth, the controller processes the packet-in message and updates the flow table by processing the packet-out message. Fifth, the switch then extracts the temporarily buffered packet from the CPU based on Buf_id in the packet-out message and outputs it to the destination.

The internal buffering for software-based SDN switches can be easily realised by configuring packet-in events to support buffering of packets. However, for hardware-based SDN switches, there are very few commodity switches that support internal buffering. Pica8 switches are among the few that support OpenFlow's feature to configure temporary buffering of packets [99], while other commodity switch manufacturers like Cisco [100], HP enterprise [101], Juniper [102], Arista network [103], and Extreme network [104] still do not support internal buffering. The reason behind fewer commodity switches supporting internal buffering is due to hardware limitations in hardware switches. This is also the reason why there is almost no experimental research conducted on SDN commodity switches to analyse internal buffering.

In [105] the authors adopted an SDN for wireless mesh networks and show that the delay variability and limited bandwidth over the wireless network induces throughput and packet loss. However, no internal buffering was considered. The use of internal buffering in [105] could have improved the channel utilization in the SDN-enabled wireless networks for increasing control traffic. Earlier studies [106, 1] suggest that the smoothing of control traffic via the internal buffer would reduce the losses during periods of poor wireless connectivity or sudden burst of new flows to a mesh router. However, these studies have not explored the drawbacks of internal buffering in an SDN.

For SDWAN (i.e. Software Defined Wireless Access Network) applications, a multi-path OpenFlow channel for resilience and scalability in wireless environments was proposed in [107]. In SDWANs, the control path may incur failure due to many reasons, such as deep fading, mobility, etc. In such cases, buffering packets in the switch's internal buffer allows the switch to continue operating momentarily while the control channel recovers back to its stable state.

Hu *et al.* [108] take a radically different approach whereby the control packets are neither buffered nor sent to the controller immediately but sent through a looping path - inducing delay to allow the control messages to be processed and the feedback from the controller. The internal buffering in [108] could have reduced the delay at the cost of extra memory.

From a performance modelling perspective, queueing theory has been widely used to model and predict the performance of an SDN [11, 66, 67, 12, 70, 109, 69, 76, 78]. These studies use a generic model such as the one shown in Fig. 2.5 where the output buffer of a switch is modelled either as a single shared queue or two-priority queues. While none of the above mentioned models consider the internal buffering capabilities of an SDN switch, they pave the way for building a new model for internal buffering within SDN switches. The models for the internal buffering mechanisms will guide network analyst with both the benefits and the drawbacks of internal buffering in an SDN.

A summary of existing queueing models for the packet encapsulation methods at the switch is shown in Table 2.4. In Table 2.4, the first column denotes existing queueing models for SDN switches, the second column indicates the packet processing configurations for asynchronous messages, the third column denotes the queueing model in Kendall's notation and the final column denotes the type of analysis.

From Table 2.4, it is clear that none of the surveyed analytical models have considered internal buffering of data packets. The effects of internal buffering on the performance of an SDN network need to be investigated through modelling and performance analysis.

Table 2.4: Summary of queueing models for packet encapsulation methods at the SDN switch.

Model	**Processing**			**Switch**	**Analysis**	
	E	I	E-I	**model**	EXA.	APP.
Jarschel [66]	✓			M/M/1		✓
Mahmood [67]	✓			M/M/1		✓
Yen [69]	✓			M/M/1		✓
Miao [12]	✓			M/M/1		✓
Miao [70]	✓			MMAP		✓
Goto [11]	✓			GI/M/1/K	✓	
Javed [76]	✓			M/G/1		✓
Lai [78]	✓			MMPP/M/1		✓

E denotes full encapsulation of a packet with no need for internal buffering, I denotes a queueing of a data packet at a switch server that supports internal buffering, E-I denotes partial encapsulation of a packet with internal buffering, EXA. denotes exact analysis and APP. denotes approximate analysis.

2.4.3 SDN switches: Software vs. Hardware

Most of the previous works have modeled a software switch where the output buffers of the switch block in Fig. 2.5 are modelled either as a: (i) single shared queue or (ii) a two-priority queue (fast path vs. slow path).

The analytical model in [66] models the switch as a shared M/M/1 queue and the controller block modelled by an M/M/1/K queueing system respectively. In this model, the switch queue is a software switch and a fraction of incoming external traffic to the switch is forwarded to the controller. While the shared queue model simplifies analyses, it does not reflect the realities of the different packet processing time scales of hardware and software switches.

The model presented in [67] considers software switch and controller collectively as Jackson's network mimicking the model in [66]. A single server in the model in [67] does not reflect the hardware processing speeds and internal queueing structure of hardware switches. To model a hardware switch through the model in [67] will necessitate a multi-server queue in place of a single server.

The model presented in [72] has assumed that the switch service time has a two-phase hyperexponential distribution and therefore modelled as an $M/H_2/1$ queue which was studied earlier in [73]. In the work of [72], there are two different service distributions for packets arriving at the switch, one that does not have matching information and needs to be sent to the controller and other which has matching information. While both analyses do not distinguish between hardware and software switches, the model in [72] does pave the way for modelling hardware switches via the two phases of the hyperexponential distribution.

The Log-Normal Mix Model (LNMM) has been proposed for OpenFlow switch in [76] to determine the path latency. In this model, they have assumed switch as a M/G/1 model with a log-normal mixture model as the service distribution. Furthermore, M/M/1 is demonstrated as poor fit for OpenFlow switch through experiments performed on Mininet, MikroTik Routerboard 750GL and GENI but did not consider it from the perspectives of hardware and software switches. The lack of separation between the control plane and data plane packets in the model by [76] makes it a poorer fit (compared to other published works) for modelling SDN switches.

The model presented in [12] uses preemptive priority queues in a switch with infinite capacity. This work was extended into [70] which assumes arrival as a MMPP under a bursty multimedia traffic scenario. Both the work in [76] and [12] have features suitable for modelling hardware switches because it can accommodate the different processing speeds in the TCAM and the CPU. The processing speeds for the TCAM and the CPU were characterized by different distributions of service times.

To reflect a realistic OpenFlow switch more accurately, priority queues with finite capacity are used instead [11]. The priority queue in [11] is non-preemptive whereby the lower priority queue is serviced only when there are no packets in the higher priority queue. The reality of limited buffer sizes for "C" ingress ports each modelled as $M/M/1/K/\infty$ is then specifically addressed by a C-$M/M/1/K/\infty$ queueing model that has been proposed for computing the minimum buffer size requirement of an OpenFlow switch [110]. The model presented in [77] is among the first to model an SDN hardware switch using queueing theory but does not consider a hardware data plane. In this model, the switch is assumed to be an M/Geo/1 model where inter-arrival is exponentially distributed and the service time is geometrically distributed but this model does not account for the switch-controller interaction. In this work, the performance of the switch is defined as the time required by the switch to process the packets without any interaction with controller. However, the analysis in [77] does not map the workings of hardware switch such as flow matching and dedicated packet processing to the queueing model.

Based on our survey of related research presented in this section, we summarize our findings in Table 2.5. In Table 2.5, the first column denotes existing queueing models for SDN software and hardware switches, the second column denotes the queueing model in Kendall's notation while the third and fourth column denote the type of analysis (exact vs. approximation) and the switch type (hardware vs. software). It is clear that apart from [77] we have not found other works that model an SDN hardware switch.

From Table 2.5, it is clear that most of the analysis is an approximation and has assumed software switches. A unified queueing model that characterizes the performance of hardware and software switches in an SDN will allow network engineers to predict performance during their SDN deployments.

Table 2.5: Summary of queueing models for SDN switches.

Model	**Switch model**	**Analysis**		**Switch Type**	
		EXA.	APP.	SW	HW
Jarschel [66]	M/M/1		✓	✓	
Mahmood [67]	M/M/1		✓	✓	
Yen [69]	M/M/1		✓	✓	
Miao [12]	M/M/1		✓	✓	
Shang [72]	M/H_2/1		✓	✓	
Sood [77]	M/Geo/1		✓		✓
Miao [70]	MMAP		✓	✓	
Goto [11]	GI/M/1/K	✓		✓	
Javed [76]	M/G/1		✓	✓	

SW denotes software, HW denotes hardware, EXA. denotes exact analysis and APP. denotes approximate analysis.

2.5 Summary

SDN is an emerging network paradigm that requires modelling and performance analysis to identify potential bottlenecks that could hinder it's performance. An analytical approach for performance analysis is not only cost-effective but is a good choice for designing a new network and optimising the performance of an existing network architecture. Queueing theory is a popular mathematical tool that has been widely used for a predictive packet-level analysis of an SDN switch and will be the approach used in this thesis.

From the literature review, several research works have independently studied buffer sharing mechanisms but have not address the tradeoffs between those mechanisms. We address tradeoffs with the help of buffer dimensioning in Chapter 3.

Similarly, existing queueing models for an SDN have focused on switches that immediately send packets to the controller for decisioning, with no existing models investigating the impact of the internal buffer in an SDN switch and the associated tradeoffs of having an internal buffer. We propose an analytical model for an SDN switch with the internal buffer to investigate the potential benefits, drawback and trade-offs of internal buffering in Chapter 4.

Finally, existing queueing models for an SDN have focused on the performance analysis of software switches with almost none to address the tradeoffs in choosing software switch over hardware switch, and vice-versa. We develop a unified queueing model for characterizing the performance of hardware switches and software switches for an SDN in Chapter 5.

Chapter 3

Shared vs. Priority Buffer

From a generic queueing model for an SDN network discussed in Section 2.4, the output buffer of the switch block in Fig. 2.5 is modelled either as : (i) a single shared queue or (ii) a two-priority queue. In the shared queue model, packets passing through the fast path and slow path share a single queue with first in first out (FIFO) service discipline. While in the two-priority queue, fast path and slow path packets are queued separately and each queue is served in a FIFO manner without preemption. In the two-priority queue model, packets in the slow path are always served with higher priority over fast path packets but the server does not preempt service of fast path packets once it has commenced. This is referred to as a service without preemption.

3.1 Infinite and finite capacity queue models

Figure 2.5 shows that the controller block and switch block can be modelled as infinite (denoted by M/M/1) or finite (denoted by M/M/1/K) capacity queue. For the queueing network with infinite capacity queues, the stationary behaviour of the network can be obtained with the help of product-form analysis [111]. The product-form queueing analysis requires the assumption that each queue in the network is independent of others.

This assumption is not realistic as the state of one queue might affect the behaviour of another queue especially in feedback queueing network.

Similarly, the queueing network with finite capacity queues result in a non-product-form queueing network. This is due to the fact that the feedback queueing network traffic mixes with external arrivals that result in non-Poissonian inputs to the switch. Hence, the stationary behaviour for such a network cannot be obtained using product-form network analysis. The stationary behaviour for such a network can be obtained by the global-balance equations [112], however, the analysis is significantly more complex due to the large state space [97].

In this thesis, the switch is assumed to have a finite queue capacity for both shared and priority buffer to analyse the realistic behaviour of an SDN. In the following section, buffer dimensioning with queueing models is discussed to address the packet loss incurred in a finite queue capacity switch due to the blocking.

3.2 Buffer dimensioning with queueing models

A key design question for SDN switches is determining the output buffer size (K) given a desired loss probability. Buffers absorb temporary fluctuations in packet arrivals and must be dimensioned to achieve a desired loss probability that is lower than the loss probability on the outgoing links. For example, a 1Gbps outgoing link using the 802.3 standard specifications must have bit error rate (BER) below 10^{-12} [113] while a 40Gbps link must have errors below 10^{-20}. Thus, it is in the interest of the switch designer to dimension the buffer to ensure that the losses due to queueing are below the BER of the outgoing links.

To answer this design question, the buffers are first assumed to be an infinite queue, and the queue is truncated at some finite integer K such that the desired loss probability is achieved [114, 115]. To render the probability of a loss below the desired loss rate (e.g. 10^{-12} for 1Gbps link), the switch buffer may be dimensioned so that the probability of the queue in an unlimited buffer exceeding the capacity K packets at an arbitrary instant is less than the target value.

In the simplest form, the buffer space requirement K, is measured in bits, line rate is specified in bits per second, and the queue service period is specified in seconds. However, the required buffer space, in practice, is more meaningful if measured in packets. The minimum output queue capacity for switch (denoted by $K_{\min}$) can be derived using an infinite queue model. However, losses in the queue are typically expressed as Packet Error Ratio (PER) while losses in the outgoing links are expressed by BER. The relationship between BER and PER is given as

$$PER = 1 - (1 - BER)^{N_b}, \tag{3.1}$$

where N_b is the number of bits in the packet. Therefore, given a PER, the minimum number of packets the switch can queue is obtained from M/M/1 queueing model which guarantees the desired loss probability in a switch is less than PER. In an M/M/1 queue, the probability that the queue length (L) exceeds $K_{\min}$ is given by $P_r\{L > K_{\min}\} = \rho$, where ρ is the server utilization at the queue for given $K_{\min}$. Therefore, if PER is the desired loss probability, ρ should be less than PER. Thus, a queue with a capacity of at least $K_{\min}$ is sufficient to ensure the loss probability that is bounded below PER. The value of $K_{\min}$ is calculated as

$$K_{\min} \geq \frac{\log[PER]}{\log[\rho]}. \tag{3.2}$$

In this thesis, the minimum buffer capacities of queues are approximated values and do not affect the comparison of different models. The approximated buffer helps in identifying trends while performing sensitivity analysis.

3.3 SDN switch design guidelines through comparative analysis

To study the effect of buffer sharing in an SDN switch, two queueing models: Model SE and Model SPE are defined. These models are based on different queue structures in the switch. In Models SE and SPE, "S" refers to the switch with software data plane, "E" refers to the encapsulation method in switch where packets are encapsulated and forwarded to the controller, and "P" refers to the priority queue in data plane. The term software data plane makes it explicit that the switch runs forwarding software (e.g. OpenVswitch, Lagopus or DPDK) on CPU (central processing unit) rather than hardware mircocoded forwarding engines, thus a single server model is appropriate. Model SE uses a single queue for the fast path and slow path [66, 67], this is also referred to as the Shared Buffer Queue model. Model SPE uses priority queues as in [11, 12, 70], and is also referred as Priority Queue Model. For brevity, Model SE is denoted as SE and Model SPE as SPE.

By comparing SE and SPE, two switch design questions are answered that were mentioned earlier in Section 1.3 as research objectives for buffer sharing models:

- SE vs. SPE: Under what conditions, shared buffer queue structure (as in SE) can be used instead of priority buffer queues (as in SPE).
- Buffer dimensioning: Compare the minimum required buffer space between SE and SPE given a desired loss probability (e.g. dictated by outgoing link on switch).

For performance analysis of SE and SPE, the controller is assumed to have an infinite capacity queue. The external arrival at the switch is assumed to be Poisson and is denoted by λ_1, the service rate of the switch is denoted by μ_{sp}, and the service rate of the controller is denoted by μ_c. If an incoming data packet has no matching FTE or table miss, the packet is sent to the controller, and this occurs with probability referred as table miss probability represented as β.

The average time to install FTEs in an SDN switch is the primary performance metric used for comparing SE and SPE. It is commonly denoted by mean sojourn time of packet in queueing theory. Also, when using Little's theorem (i.e. mean sojourn time of packet is ratio of average packet length and arrival rate), arrival rate is replaced by throughput (denoted by T) for the finite buffer in the switch due to the packet losses.

3.3.1 Model SE: an SDN switch with a single shared queue

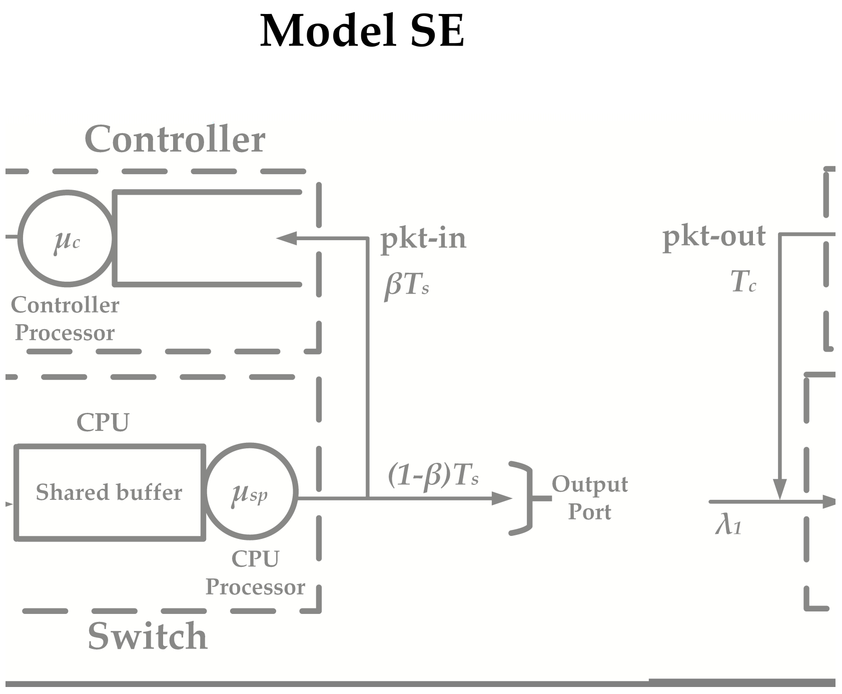


Figure 3.1: Model SE – an SDN Switch with a single shared queue.

Model SE uses a single queue for the switch shared between packets traversing the fast path and slow path. Figure 3.1 shows SE with a finite capacity queue at switch and the net traffic entering the switch is a sum of external traffic and feedback dependent on the controller (thus depicted as a single arrow). As such, the queue in the switch is denoted as GI/M/1/K to represent independent arrivals with general distribution [116]. Such models have been separately analysed in [66] and [67].

The standard M/M/1 expression for the average queue length and average delay does not hold for switches modelled with finite capacity (GI/M/1/K) queue because the traffic from external arrivals mix with packets fed back from the controller resulting in the aggregate process losing its Markovian property [116, 117].

The queues for the network shown in Fig. 3.1 are modelled as a homogeneous QBD process and the matrix analytic method (MAM) is applied to derive the average queue length and average delay. The finite capacity switch in Fig. 3.1 is modelled as continuous Markov process with its state defined by $\{(n_c(t), n_s(t)), t \geq 0\}$. These state variables denoted by $n_c(t)$ and $n_s(t)$ represent the number of packets in controller and switch, respectively. Let the Markov process at time t for SE be defined as

$$\{n_c(t), n_s(t)\} = \{x, y\} \tag{3.3}$$

where $x \in \mathbb{Z}_+$ and $y \in \mathbb{Z}_+^{\leq K_{SE}}$. The K_{SE} is the minimum output buffer capacity for the switch in SE. The permissible transitions for the Markov process $\{(n_c(t), n_s(t))\}$ are listed in Table 3.1. With the help of permissible transitions given in Table 3.1 and standard methods from MAM, an infinitesimal generator matrix Q can be derived to yield the stationary state distribution (π). A detailed derivation of the generator matrix for SE is detailed in the following subsection.

3.3.1.1 Generator matrix

Here, sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the generator matrix for SE are derived.

Table 3.1: Permissible transitions for Model SE.

Event	**From**	**To**	**Rate**
One packet arrives at the switch	(x, y)	$(x, y+1)$	λ_1
One packet departs from the switch to out of the system (SE)	$(x, y>0)$	$(x, y-1)$	$\mu_{sp}(1-\beta)$
One packet forwarded from the switch to the controller	$(x, y>0)$	$(x+1, y-1)$	$\mu_{sp}\beta$
One packet serviced by the controller to the switch	$(x>0, y)$	$(x-1, y+1)$	μ_c

Elements of matrix A_0: The sub-matrix A_0 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.3)) increases by 1:

$$A_{0(y,y')} = \begin{cases} \mu_{sp}\beta, & y' = y-1, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix A_1: The sub-matrix A_1 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.3)) remains unchanged:

$$A_{1(y,y')} = \begin{cases} \mu_{sp}(1-\beta), & y' = y-1, \\ \lambda_1, & y' = y+1, \\ 0, & \text{otherwise.} \end{cases}$$

The diagonal elements of $A_{1(y,y')}$ where y is equal to y' has the following cases:

$$A_{1(y,y')} = \begin{cases} -\lambda_1 - \mu_c, & y = 0, \\ -\lambda_1 - \mu_c - \mu_{sp}, & 0 < y < K_{SE}, \\ -\mu_{sp}, & y = K_{SE}, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix* B_1*: The sub-matrix B_1 represents the phase distribution of controller and switch when the number of packets in the controller remains unchanged and there is no packet in the controller (i.e. $n_c(t)$ or x in Eq. (3.3) is equal to 0). The sub-matrix B_1 is identical to A_1 except that its diagonal element is different from A_1.

$$\therefore B_1 = A_1 \text{ (for } \mu_c\text{=0).}$$

Elements of matrix* A_2*: The sub-matrix A_2 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.3)) decreases by 1:

$$A_{2(y,y')} = \begin{cases} \mu_c, & y' = y + 1, \\ 0, & \text{otherwise.} \end{cases}$$

3.3.1.2 Network performance metrics for SE

The sub-matrices A_0, A_1, B_1, and A_2 are input to MAM which will output π distribution for SE. This π distribution is used to compute performance metrics such as throughput, average queue length, and packet loss probability. The throughput and average queue length of the switch are used to compute the average time to install FTEs in the shared buffer.

Throughput of the switch (denoted by T_s) is given by the sum of probabilities that the switch has at least one data packet to forward (with service rate of μ_{sp}) and this is given by

$$T_s = \mu_{sp} \sum_{x=0}^{\infty} \sum_{y=1}^{K_{SE}} \pi_{x,y}, \tag{3.4}$$

where $\pi_{x,y}$ is the stationary probability for x packets in controller and y packets in switch.

Throughput of the controller (denoted by T_c) is given by the sum of probabilities that the controller has at least one control packet to forward (with service rate of μ_c) and this is given by

$$T_c = \mu_c \sum_{x=1}^{\infty} \sum_{y=0}^{K_{SE}} \pi_{x,y}. \tag{3.5}$$

Average number of data packets (denoted by L_{SE}) is the average number of data packets in SE where data packets travel through the switch and the controller. Therefore, L_{SE} is expressed as:

$$L_{SE} = \sum_{x=0}^{\infty} \sum_{y=0}^{K_{SE}} (x+y)\pi_{x,y}. \tag{3.6}$$

Average queue length of switch (denoted by L_s) is the average number of packets in the switch expressed as

$$L_s = \sum_{y=1}^{K_{SE}} \pi_y \times y, \tag{3.7}$$

where π_y is the marginal probability for the average number of packets in the switch, and this marginal probability is defined as

$$\pi_y = \sum_{x=0}^{\infty} \pi(x,y). \tag{3.8}$$

Average data packet transfer delay (denoted by t_{SE}) in the mean sojourn time of a data packet in SE. It is obtained by applying Little's theorem to Eq. (3.6) which is expressed as:

$$t_{SE} = L_{SE}/T_{SE}, \tag{3.9}$$

where T_{SE} is the throughput of SE expressed as:

$$T_{SE} = (1 - \beta)T_s. \tag{3.10}$$

Average time to install FTEs (denoted by tt_{SE}) at the switch is the average packet transfer delay (commonly denoted by the mean sojourn time of the packet). Applying Little's theorem to Eq. (3.7) yields the average time to install FTEs for SE which is expressed as

$$tt_{SE} = L_s/T_s, \tag{3.11}$$

where tt_{SE} is the average time to install an FTE into the switch flow table for SE.

Packet loss probability (denoted by PL_{SE}) is the average number of packets being blocked or dropped in the shared buffer out of total incoming packets. It is expressed as

$$PL_{SE} = 1 - T_{SE}/\lambda_1. \tag{3.12}$$

Note that the validity of the expressions given in this section are contingent on the stability condition for the queueing network. Following standard M/M/1 theory (also given in [66]), the queues are stable if the following inequality holds:

$$\beta\lambda_1 - (1 - \beta)\mu_c < 0. \tag{3.13}$$

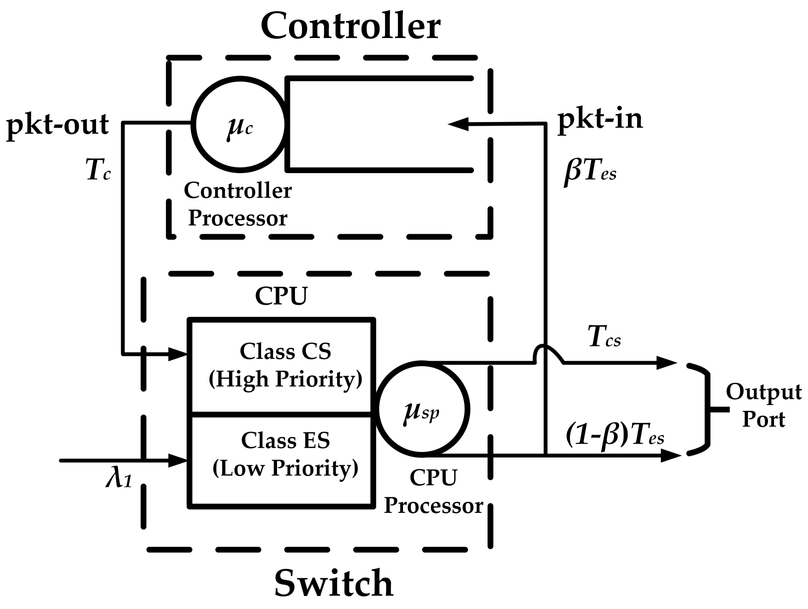


Figure 3.2: Model SPE – an SDN switch with a two-priority queue.

3.3.2 Model SPE: an SDN Switch with a two-priority queue

Figure 3.2 shows the model that uses priority queues for the switch with a finite capacity. The priority queues provides isolation between packets arriving from the controller and external packets arriving at the switch. In this model, non-preemptive priority queues are used in which the lower priority queue is serviced when there are no packets in the higher priority queue. The two priority classes proposed in [11] reflect the realities of the fast path and slow path, these priorities are called Class CS (control traffic from controller to switch), and Class ES (external data traffic arrival at switch) to indicate the different packet processing paths.

Class ES represents the class for an external data packet arrival to the switch. If the external data packet has no matching entry in the switch, the packet is sent to the controller, and this occurs with probability β. Class CS represents the class for packets fed back to the switch from the controller and must be forwarded out to the destination. Packets in the Class ES already have a forwarding rule installed in the switch and are assigned to the lower priority with respect to the Class CS packets. Both Class CS and Class ES queues shares service rate μ_{sp}.

The SPE queues are modelled with a homogeneous QBD process with indexed by three state variables $\{(n_c(t), n_{cs}(t), n_{es}(t)), t \geq 0\}$. The state variables denoted by $n_c(t)$, $n_{cs}(t)$, and $n_{es}(t)$ represent the number of packets in controller, Class CS, and Class ES, respectively. The minimum output buffer capacity for the switch in SPE is equal to K_{SPE}. It is assumed that the Class CS and Class ES has the queue capacity of K_1 and K_2 respectively. Let the Markov process at time t for SPE be defined as

$$\{n_c(t), n_{cs}(t), n_{es}(t)\} = \{x, y, z\} \tag{3.14}$$

where $x \in \mathbb{Z}_{+}$, $y \in \mathbb{Z}_{+}^{\leq K_1}$ and $z \in \mathbb{Z}_{+}^{\leq K_2}$. The permissible transitions for the Markov chain $\{(n_c(t), n_{cs}(t), n_{es}(t))\}$ are shown in Table 3.2 and these help us determine the stationary distribution probability (π) for SPE.

3.3.2.1 Generator matrix

Here, sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the generator matrix for SPE are derived.

Elements of matrix A_0**:** The sub-matrix A_0 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.14)) increases by 1:

$$A_{0(y,y')} = \begin{cases} \tilde{A_0}^{(\mathrm{y})}, & y' = y = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where

$$\tilde{A_0}^{(0)}{}_{(z,z')} = \begin{cases} \mu_{sp}\beta, & z' = z - 1, \\ 0, & \text{otherwise}. \end{cases} \tag{3.15}$$

Table 3.2: Permissible transitions for Model SPE.

Event	From	To	Rate
One packet departs from the switch out of the system (SPE)	$(x, 0, z > 0)$	$(x, 0, z-1)$	$\mu_{sp}(1-\beta)$
One packet departs from the Class CS out of the system (SPE)	$(x, y > 0, z)$	$(x, y-1, z)$	μ_{sp}
One packet arrives at the Class ES	(x, y, z)	$(x, y, z+1)$	λ_1
One packet forwarded from the Class ES to the controller	$(x, 0, z > 0)$	$(x+1, 0, z-1)$	$\mu_{sp}\beta$
One packet serviced by the controller to Class CS	$(x > 0, y, z)$	$(x-1, y+1, z)$	μ_c

Elements of matrix A_1**:** The sub-matrix A_1 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.14)) remains unchanged:

$$A_{1(y,y')} = \begin{cases} {A_{11}}^{(\mathrm{y})}, & y' = y, \\ {A_{12}}^{(\mathrm{y})}, & y' = y-1, \\ 0, & \text{otherwise}, \end{cases} \tag{3.16}$$

where

$${A_{11}}^{(\mathrm{y})}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z+1, \\ \mu_{sp}(1-\beta), & y = 0, \quad z' = z-1, \\ 0, & \text{otherwise}, \end{cases} \tag{3.17}$$

and

$${A_{12}}^{(\mathrm{y})}{}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z, \\ 0, & \text{otherwise}. \end{cases} \tag{3.18}$$

The diagonal elements of ${A_{11}}^{(\text{y})}{}_{(z,z')}$ where z is equal to z' has the following cases:

$$
{A_{11}}^{(\text{y})}{}_{(z,z)} = \begin{cases} -\lambda_1 - \mu_c, & y = 0, \quad z = 0, \\ -\lambda_1 - \mu_c - \mu_{sp}, & y = 0, \quad 0 < z < K_2, \\ -\lambda_1 - \mu_c - \mu_{sp}, & 0 < y \leq K_1, \quad 0 \leq z < K_2, \\ -\mu_{sp}, & 0 \leq y \leq K_1, \quad z = K_2, \\ 0, & \text{otherwise.} \end{cases} \tag{3.19}
$$

Elements of matrix* B_1*: The sub-matrix B_1 represents the phase distribution of controller and switch when the number of packets in the controller remains unchanged and there is no packet in the controller (i.e. $n_c(t)$ or x in Eq. (3.14) is equal to 0). The sub-matrix B_1 is identical to A_1 except that its diagonal element is different from A_1.

$$\therefore B_1 = A_1 \text{ (for } \mu_c\text{=0).}$$

Elements of matrix* A_2*: The sub-matrix A_2 represents the phase distribution of controller and switch when the number of packets in the controller (i.e. $n_c(t)$ or x in Eq. (3.14)) decreases by 1:

$$
A_{2(y,y')} = \begin{cases} {A_{20}}^{(\text{y})}, & y' = y + 1, \\ 0, & \text{otherwise,} \end{cases} \tag{3.20}
$$

where

$$
{A_{20}}^{(\text{y})}{}_{(z,z')} = \begin{cases} \mu_c, & z' = z, \\ 0, & \text{otherwise.} \end{cases} \tag{3.21}
$$

3.3.2.2 Network performance metrics for SPE

The sub-matrices A_0, A_1, B_1, and A_2 are input to MAM which will output π distribution for SPE. This π distribution is used to compute the average time to install FTEs and packet loss probability in the priority buffer. These performance metrics are computed using throughputs and average queue lengths of Class ES and Class CS.

Throughput of the Class ES (denoted by T_{es}) is given by the sum of probabilities that the Class ES has at least one data packet to forward (with service rate of μ_{sp}) and no packet in the Class CS in the stationary state, and this is given by

$$T_{es} = \mu_{sp} \sum_{x=0}^{\infty} \sum_{z=1}^{K_2} \pi_{x,0,z}. \tag{3.22}$$

Throughput of the Class CS (denoted by T_{cs}) is given by the sum of probabilities that the Class CS has at least one data packet to forward (with service rate of μ_{sp}) and this is given by

$$T_{cs} = \mu_{sp} \sum_{x=0}^{\infty} \sum_{y=1}^{K_1} \sum_{z=0}^{K_2} \pi_{x,y,z}. \tag{3.23}$$

Throughput of the controller (denoted by T_c) is given by the sum of probabilities that the controller has at least one control packet to forward (with service rate of μ_c) and this is given by

$$T_c = \mu_c \sum_{x=1}^{\infty} \sum_{y=z}^{K_1} \sum_{z=0}^{K_2} \pi_{x,y,z}. \tag{3.24}$$

Average queue length of the Class ES (denoted by L_{es}) is the average number of packets in the Class ES of switch which is expressed as

$$L_{es} = \sum_{z=1}^{K_2} \pi_z \times z, \tag{3.25}$$

where π_z is the marginal probability for average number of packets in the Class ES of switch and is expressed as

$$\pi_z = \sum_{x=0}^{\infty} \sum_{y=0}^{K_1} \pi(x, y, z). \tag{3.26}$$

Average queue length of the Class CS (denoted by L_{cs}) is the average number of packets in the Class CS of the switch which is expressed as

$$L_{cs} = \sum_{y=1}^{K_1} \pi_y \times y, \tag{3.27}$$

where π_y is the marginal probability for average number of packets in the Class CS of the switch and is expressed as

$$\pi_y = \sum_{x=0}^{\infty} \sum_{z=0}^{K_2} \pi(x, y, z). \tag{3.28}$$

Average number of data packets (denoted by L_{SPE}) is the average number of data packets in SPE where data packets travel through the switch (the Class ES and the Class CS) and the controller. Therefore, L_{SPE} is expressed as:

$$L_{SPE} = \sum_{x=0}^{\infty} \sum_{y=0}^{K_1} \sum_{z=0}^{K_2} (x + y + z) \pi_{x,y,z}. \tag{3.29}$$

Average data packet transfer delay (denoted by t_{SPE}) in the mean sojourn time of a data packet in SPE. It is obtained by applying Little's theorem to Eq. (3.29) which is expressed as:

$$t_{SPE} = L_{SPE} / T_{SPE}, \tag{3.30}$$

where T_{SPE} is the throughput of SE expressed as:

$$T_{SPE} = T_{cs} + (1 - \beta) T_{es}. \tag{3.31}$$

Average time to install FTEs (denoted by tt_{SPE}) is the average packet transfer delay at the Class CS of the switch. Applying Little's theorem on Eq. (3.27) yields the average delay for the packet that pass through the Class CS which is expressed as

$$tt_{SPE} = L_{cs}/T_{cs}. \tag{3.32}$$

where tt_{SPE} is the average time to install an FTE into the finite queue capacity switch for SPE.

Packet loss probability (denoted by PL_{SPE}) is the average number of packets being blocked or dropped in the priority buffer out of total incoming packets. It is expressed as

$$PL_{SPE} = 1 - T_{SPE}/\lambda_1. \tag{3.33}$$

The validity of the expressions given for SPE are contingent on the following stability condition:

$$\beta\lambda_1 - \mu_c < 0. \tag{3.34}$$

The proof is given in [11].

3.3.3 Buffer dimensioning: SE vs. SPE

For the buffer dimensioning problem in SE, the switch queue is assumed as M/M/1 (see Section 3.2). The minimum capacity for the switch in SE i.e. K_{SE} is calculated using Eq. (3.2) as

$$K_{SE} \geq \frac{\log[PER]}{\log[\rho_s]}, \tag{3.35}$$

where ρ_s is the server utilization at the switch defined as $\rho_s = (1+\beta)\lambda_1/\mu_{sp}$.

Likewise, for the buffer dimensioning of SPE, the switch queues are assumed to be M/M/1 and the PER is fixed. The minimum queue capacities for Class CS and Class ES of the switch in SPE are denoted as K_1 and

K_2 respectively and can be calculated using Eq. (3.2) as:

$$K_1 \geq \frac{\log[PER]}{\log[\rho_{cs}]}, \qquad K_2 \geq \frac{\log[PER]}{\log[\rho_{es}]}, \tag{3.36}$$

where ρ_{cs} and ρ_{es} are the server utilization at the Class CS and Class ES of the switch respectively. These server utilization (ρ_{cs} and ρ_{es}) are defined as:

$$\rho_{cs} = \frac{\beta \lambda_1}{\mu_{sp}}, \qquad \rho_{es} = \frac{\lambda_1}{\mu_{sp}}.$$

Therefore, the minimum total queue capacity of switch in SPE is the sum of minimum queue capacity for Class CS and Class ES which is given as

$$K_{SPE} = K_1 + K_2. \tag{3.37}$$

3.4 Results

The parameters used for numerical evaluation are shown in Table 3.3. From Table 3.3, the table miss probability β varies from 0.1 to 1, the external arrival rate (λ_1) is set of 120 or 240 or 480 packets/sec that typify arrivals from a small business premise [118] to a campus area network [119, 120]. The switch processing rate (μ_{sp}) is assumed to be 1000 packets/sec which is greater than λ_1 to assure stationary distribution and an Ethernet network is assumed for which the BER is assumed to be 10^{-12}. The Transmission Control Protocol (TCP) is used as the transport protocol with maximum transmission unit (MTU) of 1500 bytes. Using this value of MTU, the number of bits in the packet (N_b) is computed for Packet Error Ratio (PER) in Eq. (3.1). Thus, the PER for BER of 10^{-12} and MTU of 1500 bytes is 1.2×10^{-8} (using Eq. (3.1)). This value of PER is used to determine the minimum buffer capacity for the switch queues as discussed in Section 3.3.3.

Along with the analytical results, the simulation results obtained through a discrete event simulation of the SE and SPE queueing networks are shown.

The discrete event simulation in this thesis is based on Monte Carlo simulation where Monte Carlo simulators use random number generators to simulate the system. The events for the Monte Carlo simulation are related to the transition rates of queueing models which cause models to change their states. The Monte Carlo simulations for model SE and model SPE with pseudo codes are explained in Appendix C and Appendix D, respectively. The simulations are repeated 100 times and the 95% confidence intervals (CI) are computed on the basis that the errors are normally distributed. The analytical and simulation results are obtained for parameters β, λ_1, and μ_c that satisfy the stability conditions for both SE and SPE (Eq. (3.13) and Eq. (3.34)).

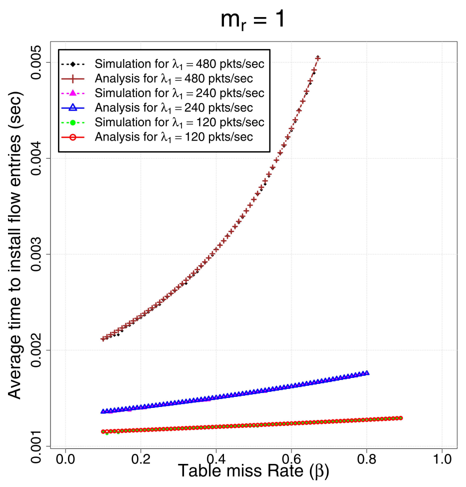
Table 3.3: Parameter used for analysis and simulation for both SE and SPE.

Parameter	**Value**
Table miss probability, β	0.1$\sim$1
Switch processing rate, μ_{sp} (packets/sec)	1000
Arrival rate, λ_1 (packets/sec)	120, 240, 480
Controller to CPU Processing Ratio, m_r	0.1$\sim$2
Bit Error Rate, (BER)	10^{-12}
MTU TCP packet size (byte)	1500

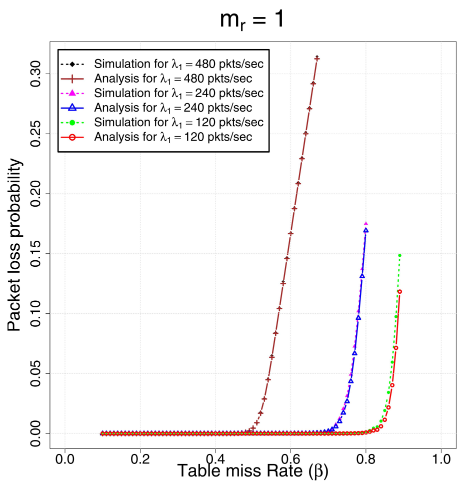
Note that the switch processing rate is assumed to be constant and controller processing rate is determined by controller to CPU processing ratio (i.e. $m_r = \mu_c/\mu_{sp}$). Also, the term "saturated switch" is introduced to refer to the net traffic arrivals to the switch which is greater than 60% of switch service rate (i.e. $0.6 \times \mu_{sp}$).

3.4.1 Validating queueing models SE and SPE

In this section, analytical results for SE and SPE are validated by comparing it with discrete event simulation result. Fig. 3.3 and Fig. 3.4 shows the validation results for SE and SPE respectively for increasing β and $m_r = 1$.

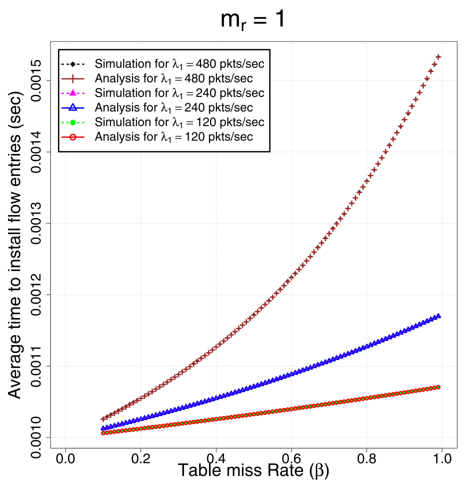


(a)

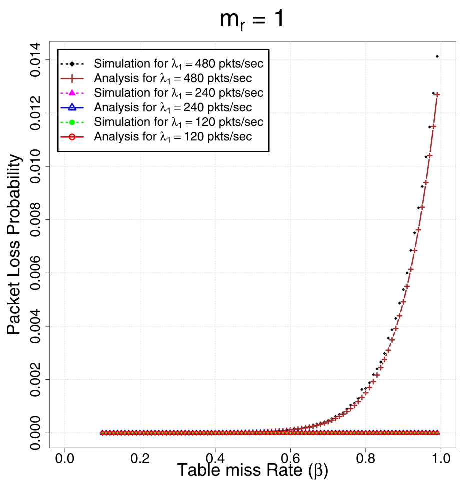


(b)

Figure 3.3: Validation of Model SE in terms of (a) Average time to install FTE, and (b) Packet loss probability.



(a)



(b)

Figure 3.4: Validation of Model SPE in terms of (a) Average time to install FTE, and (b) Packet loss probability.

To induce packet losses, the queue capacity of switch in both SE and SPE is truncated to 0.8 times of the queue capacity determined via buffer dimensioning.

For the average time to install FTEs and packet loss probability, both SE and SPE predictions from the queueing models track the simulation models. The predictions for the average time to install FTEs fall within 95% CI (Fig. 3.3a and Fig. 3.4a) and drifts out of 95% CI for packet loss probability (Fig. 3.3b and Fig. 3.4b) as β progresses past 0.8.

The error percentage between analysis and simulation predictions for the average time to install FTEs in SE is up to 2.3%, while for SPE, the error percentage is below 1%. Similarly, the error percentage for the packet loss probability in both SE and SPE is below 2.8%.

3.4.2 Switch design considerations

From Section 3.3, the analysis in this chapter answers two design questions by comparing SE and SPE:

(i) what is the tradeoff between providing traffic isolation and minimum buffer space for a desired loss probability?

(ii) does priority queueing improve the time to install flow table in an SDN switch?

The two performance measures are used to compare the performance of SE and SPE namely the relative minimum capacity and relative time to install FTE.

3.4.2.1 Relative minimum capacity

In this section, the minimum queue capacity of the switch between SE (denoted by K_{SE} as in Eq. (3.35)) and SPE (denoted by K_{SPE} as in Eq. (3.37)) is compared. In this comparison, the tradeoff between shared buffer and priority queueing buffer in an SDN switch is investigated.

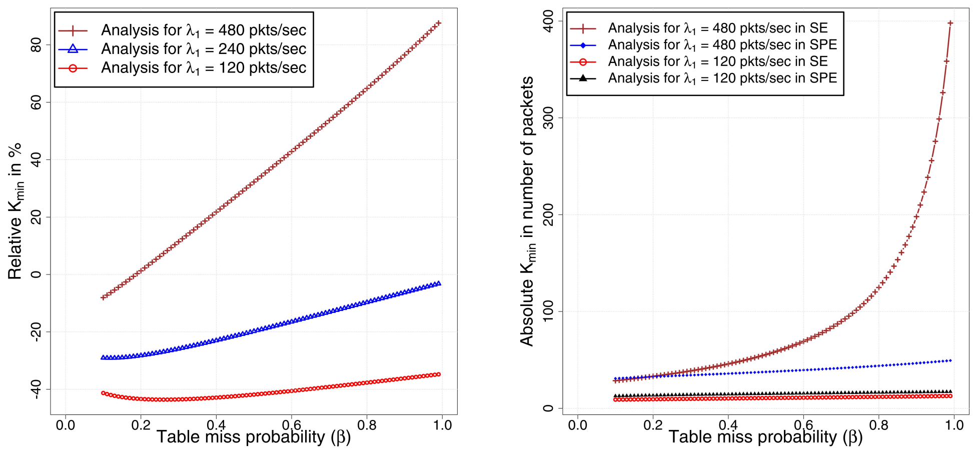


Figure 3.5: Minimum switch queue capacity (K_{min}) between SE and SPE for $\mu_{sp} = 1000$ pkts/sec and increasing β: (a) Relative difference i.e. ϵ_K and (b) Absolute value.

The relative minimum queue capacity of the switch between SE and SPE is denoted as ϵ_K which is expressed as

$$\epsilon_K = \frac{K_{\text{SE}} - K_{\text{SPE}}}{K_{\text{SE}}} \times 100\%.$$

The relative minimum queue capacity reflects the percentage difference in minimum queue space to achieve the desired loss probability. Figure 3.5a shows the ϵ_K curves and Fig. 3.5b shows the absolute values of K_{SE} and K_{SPE} for increasing β. A negative value of ϵ_K means SPE requires more capacity than SE, while a positive value implies the opposite.

For a given loss probability, SE requires approximately up to 80% more capacity than SPE. This observation is due to the increased control traffic with β which requires a larger buffer to achieve the desired loss probability. The additional capacity required for a control traffic in the Class CS is very small compared to that required for Class ES which results in the total queue capacity for Model SE being higher than Model SPE. In the non-saturated switch, the SPE requires more capacity than SE as seen in Fig. 3.5a because the Class CS queue and Class ES queue must individually be dimensioned to achieve the desired loss probability. This results in needing a larger buffer capacity than SE in the non-saturated switch.

In brief, adopting a priority queue for better isolation between the Class ES and Class CS traffic requires a smaller switch queue capacity in saturated switches, and conversely requires a larger switch queue capacity for non-saturated switches. This conclusion helps answer the first switch design question posed in Section 3.3.

3.4.2.2 Relative average time to install FTE

In this section, the average time to install FTEs between SE (denoted by tt_{SE} as in Eq. (3.11)) and SPE (denoted by tt_{SPE} as in Eq. (3.32)) is compared. In this comparison, the effects of priority queueing in the switch is investigated.

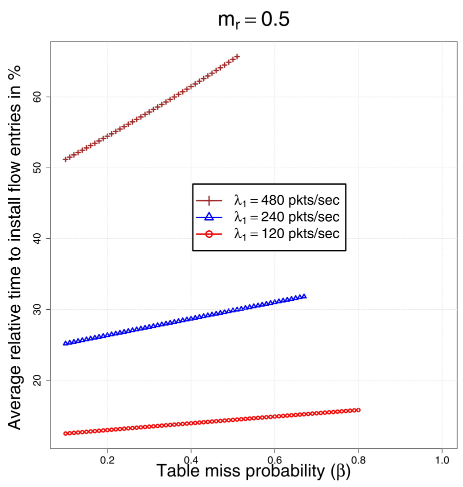
The relative time difference (denoted by ϵ_{tt}) to install FTE in the switch between SE and SPE (both with finite capacity) is calculated as:

$$\epsilon_{tt} = \frac{(tt_{SE} - tt_{SPE})}{tt_{SE}} \times 100\%.$$

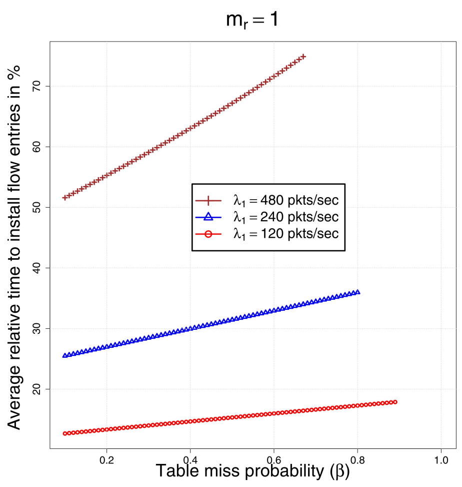
A positive value of ϵ_t means SPE has lower average time to install FTEs compared to SE.

Fig. 3.6 shows the relative average time to install FTEs in the switch between SE and SPE for increasing β and m_r for μ_{sp} of 1000 packets/sec. From Fig. 3.6, SPE exhibits up to 85% reduction in the average time to install FTEs compared to SE for increasing β and m_r. This is because the separate high priority queue for control traffic prioritizes control traffic and reduces the waiting time of control packets compared to SE where the control traffic shares the buffer with data traffic.

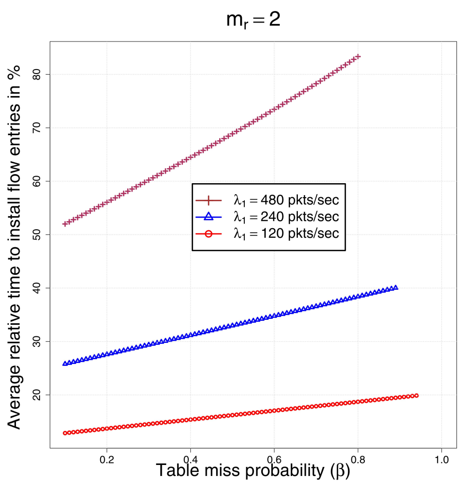
As m_r increases from 0.5 to 2, the relative time to install FTEs in SPE decreases, especially for higher β as seen in Fig. 3.6. This is because higher



(a)



(b)

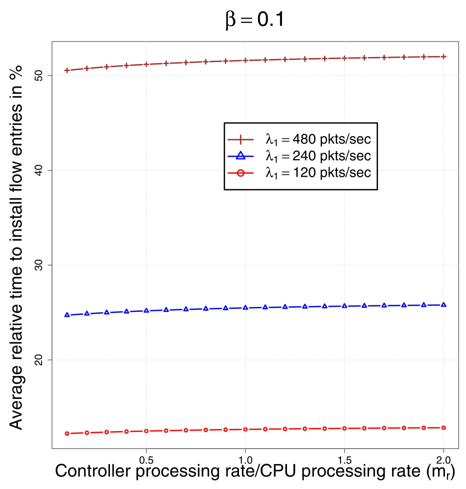


(c)

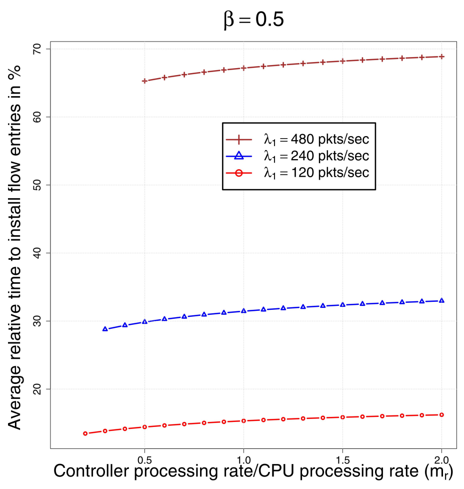
Figure 3.6: Relative time to install FTEs between SE and SPE i.e. ϵ_t for increasing β : (a) $m_r = 0.5$; (b) $m_r = 1.0$; (c) $m_r = 2.0$.

m_r represents faster processing at the controller. With a dedicated queue to serve the faster controller, the average time to install FTEs decreases for SPE while there is no change for SE.

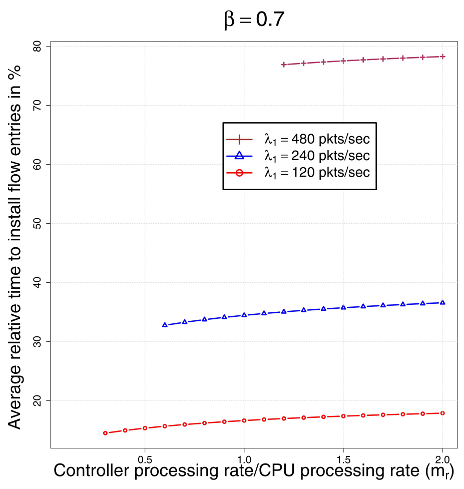
It is observed in Fig. 3.7 that as β increases from 0.1 to 1 (left to right),



(a)



(b)



(c)

Figure 3.7: Relative time to install FTEs between SE and SPE i.e. ϵ_{tt} for increasing m_r : (a) $\beta = 0.1$; (b) $\beta = 0.5$; (c) $\beta = 1$.

the relative time to install FTE increases from 50% to 78% for a saturated switch (i.e. $\lambda_1 = 480$ packet/sec). For Fig. 3.7(a) where $\beta = 0.1$, a lower average time to install FTE is predicted for SPE because a small amount of control traffic is forwarded to the controller, and increasing controller server capacity makes negligible difference to the time to install FTEs.

Overall, a switch buffer designed with the priority queueing significantly reduces the average time to install FTEs compared to a switch designed with the single shared queue. This answers the second switch design question posed in Section 3.3.

3.5 Conclusion

In this chapter, by comparing the shared buffer model (SE) and priority queueing buffer model (SPE), three key switch design questions in an SDN are answered, viz. buffer dimensioning, selection of priority or non-priority queue, and controller server capacity. The findings from the investigation are as follows:

(i) for the switch with slower processing capacity, the average time to install FTEs with SPE model decreases by up to 85% and requires up to 82% less switch buffer capacity compared to SE model,

(ii) for the faster switch processing capacity, the average time to install FTEs with the SPE model decreases as well but at the cost of higher switch buffer capacity compared to the SE model, and

(iii) for a switch with priority queues, a faster controller (i.e. $m_r \geq 1$) significantly improves relative average time to install FTEs as the table miss probability increases.

3.6 Summary

Based on the investigation from this chapter, it is evident that the priority queueing buffer model provides better performance in terms of shorter delays when installing FTEs in a switch. Therefore, the priority queueing buffer model is the preferred choice for switch designers and network analysers in an SDN as it accurately represents SDN behaviour. In the following chapters, the two-priority queueing structure will be assumed for a software data plane or CPU in an SDN switch.

Chapter 4

Internal Buffering in SDN Switches

Internal buffering in the computer communication network is a temporary buffering of packets within the switch. It has been used in traditional switches such as ATM and Banyan switches to avoid the delay and loss of packets under a heavy traffic condition [121].

Some of the benefits of internal buffering in SDN switches are: forwarding delay of data packets can be decreased [106], Quality of Service (QoS) can be improved with reduced packet loss [122], and bandwidth of the control channel can be optimized [1]. However, it will be increasingly important for the next generation of SDN switches to support internal buffering with increasing diversification of SDN deployments. There may be an intermittent connectivity between the SDN switch and the controller during SDN deployments in domains such as SDWANs, mobile SDN and IoT.

Most existing research in the literature analyses the performance of an SDN switch with no internal buffering. This is perhaps attributed to the evolving nature of OpenFlow specifications which in its current incarnation leaves the buffering of a data packet an optional feature.

In this chapter, queueing theory is used to derive a first order estimate of an OpenFlow switch's performance and to identify potential trade-offs between switch designs with the internal buffer and without the internal buffer. Queueing models are useful in predicting switch performance trends as parametrized functions and link the cause to effect relationships of the switch performance. The two main objectives of the research presented in this chapter are: (a) to model an SDN switch with the internal buffer, and (b) to investigate the effect of internal buffering in the performance of an SDN switch by comparing the queueing models for an SDN switch with and without the internal buffer, and hence identify trade-offs.

In the following section, the queueing model for an SDN switch with the internal buffer is developed and described with details of the generator matrix and performance metrics.

4.1 Model SPI: an SDN Switch with the internal buffer

Using similar convention with SPE in Section 3.3.2, the queueing model for a switch with the internal buffer is named Model SPI, where "I" refers to queueing of data packets in the internal buffer. As seen in Fig.4.1, the switch has the internal buffer for buffering of packets destined for the controller. The input buffer of the switch is modelled as a finite capacity with two-priority class queues, Class ES (for external data packets) and Class CS (for control packets) like SPE.

Model SPI consists of characterizing the four steps of packet processing as shown in Fig. 4.1: (1) external data packets arrive at the Class ES queue of the switch, (2) **if** the switch does not have a matching FTE **then** data packets are temporarily buffered in the internal memory. The CPU then forward a fraction of the data packet (around 20%) to the controller encapsulated within packet-in messages. Similarly, **if** the switch have a matching FTE **then** data packets are successfully forwarded to the desti-

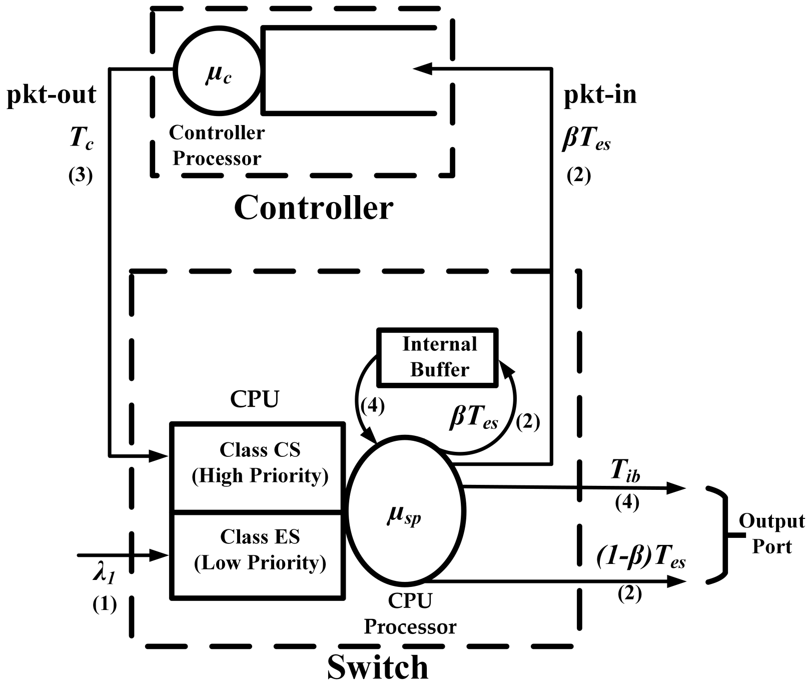


Figure 4.1: Model SPI – an SDN switch with a priority queue and the internal buffer.

nation through an output port, (3) the controller feedbacks the forwarding information with a packet-out message to Class CS of the switch, (4) and the switch processes the control packets in Class CS, updates the flow table with forwarding information, the temporarily buffered data packet is extracted from the internal buffer and forwarded to the destination through an output port.

SPI is modelled as a continuous time Markov process with four state variables, $\{(n_b(t), n_c(t), n_{cs}(t), n_{es}(t)), t \geq 0\}$. The state variables denoted by $n_b(t)$, $n_c(t)$, $n_{cs}(t)$, and $n_{es}(t)$ represent the number of packets in the internal buffer, controller, Class CS, and Class ES respectively. The minimum output buffer capacity for the switch in SPI is equal to K_{SPI}. It is assumed that the Class CS, the Class ES, and the internal buffer have the queue capacity of K_1, K_2, and K_3 respectively. Let the Markov process at time t be defined as:

$$\{n_b(t), n_c(t), n_{cs}(t), n_{es}(t)\} = \{w, x, y, z\} \tag{4.1}$$

where $w \in \mathbb{Z}_+^{\leq K_3}$, $x \in \mathbb{Z}_+$, $y \in \mathbb{Z}_+^{\leq K_1}$ and $z \in \mathbb{Z}_+^{\leq K_2}$.

The number of packets in the controller and Class CS is dependent on the number of packets in the internal buffer. Therefore, the state space of the controller can be rewritten as $x \in \mathbb{Z}_+^{\leq w}$ and y as the fixed value equal to $(w - x)$.

For example, if the number of packets in the internal buffer at some instant *t* is 1, i.e. $n_b(t) = 1$, then the permissible state space for the controller and Class CS are $n_c(t) = \{0, 1\}$ and $n_{cs}(t) = \{1, 0\}$ respectively. Due to this dependency, the Markov process in SPI is the nonhomogenous QBD process with the number of packets in the internal buffer as a level variable [87]; while the number of packets in the controller, Class CS and Class ES are phase variables. The permissible transitions for the Markov chain $\{(n_b(t), n_c(t), n_{cs}(t), n_{es}(t))\}$ are shown in Table 4.1 and these help in deriving sub-matrices (denoted by A_0, A_1, B_1 and A_2) of transition generator matrix (Q) for SPI. These sub-matrices are inputs to the matrix geometric

Table 4.1: Permissible Transitions for Model SPI.

Event	From	To	Rate
One packet arrives to the Class ES.	(w, x, y, z)	$(w, x, y, z+1)$	λ_1
One packet departs from the Class ES to out of the system (SPI).	$(w, x, 0, z>0)$	$(w, x, 0, z-1)$	$\mu_{sp}(1-\beta)$
One packet departs from the Class ES to the internal buffer and subsequently one packet-in message is sent to controller.	$(w, x, 0, z>0)$	$(w+1, x+1, 0, z-1)$	$\mu_{sp}\beta$
One packet-out serviced by the controller to the Class CS.	$(w, x>0, y, z)$	$(w, x-1, y+1, z)$	μ_c
One packet in the Class CS is processed and subsequently one packet departs from the internal buffer to out of the system (SPI).	$(w>0, x, y>0, z)$	$(w-1, x, y-1, z)$	μ_{sp}

solution for computing the stationary distribution probability (π) which is used to determine performance metrics for SPI.

4.1.1 Generator matrix

Here, sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the generator matrix for SPI are derived.

Elements of matrix A_0***:*** The sub-matrix A_0 represents the phase distribution of controller, Class CS, and Class ES when the number of packets in the internal buffer (i.e. $n_b(t)$ or w in Eq. (4.1)) increases by 1:

$$A_{0(x,x')} = \begin{cases} {A_{00}}^{(x)}, & x' = x + 1, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$${A_{00}}^{(x)}{}_{(y,y')} = \begin{cases} {A_{001}}^{(y)}, & y' = y = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$${A_{001}}^{(0)}{}_{(z,z')} = \begin{cases} \mu_{sp}\beta, & z' = z - 1, \\ 0, & \text{otherwise}. \end{cases}$$

Elements of matrix A_1***:*** The sub-matrix A_1 represents the phase distribution of controller, Class CS, and Class ES when the number of packets in the internal buffer remain unchanged and there are some packets in the internal buffer (i.e. $n_b(t)$ or w in Eq. (4.1) is a positive integer that remain unchanged):

$$A_{1(x,x')} = \begin{cases} {A_{11}}^{(x)}, & x' = x, \\ {A_{12}}^{(x)}, & x' = x - 1, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$A_{11}{}^{(x)}{}_{(y,y')} = \begin{cases} A_{111}{}^{(y)}, & y' = y, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$A_{12}{}^{(x)}{}_{(y,y')} = \begin{cases} A_{120}{}^{(y)}, & y' = y+1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$$A_{111}{}^{(y)}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z+1, \\ \mu_{sp}(1-\beta), & y = 0, z' = z-1, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$A_{120}{}^{(y)}{}_{(z,z')} = \begin{cases} \mu_c, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

The diagonal elements of $A_{111}{}^{(y)}{}_{(z,z')}$ where z is equal to z' has four distinct cases:

(i) when there is no packet in controller (i.e. $n_c(t)$ or x in Eq. (4.1) is equal to 0),

$$A_{111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_{sp}, & 0 \leq z < K_2; \\ -\mu_{sp}, & z = K_2; \\ 0, & \text{otherwise,} \end{cases}$$

(ii) when the number of packets in controller is less than the number of packets in the internal buffer i.e. $0 < x < w$ and $w < K_3$,

$$A_{111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_{sp} - \mu_c, & 0 \leq z < K_2; \\ -\mu_{sp} - \mu_c, & z = K_2; \\ 0, & \text{otherwise,} \end{cases}$$

(iii) when the number of packets in controller is equal to that in the internal buffer which is not full i.e. $x = w$ and $w < K_3$,

$$A_{111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_c, & z = 0; \\ -\lambda_1 - \mu_{sp} - \mu_c, & 0 < z < K_2; \\ -\mu_{sp} - \mu_c, & z = K_2; \\ 0, & \text{otherwise}, \end{cases}$$

(iv) when the number of packets in the controller and the internal buffer are equal to the queue size of the internal buffer i.e. $x = w = K_3$,

$$A_{111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_c, & z = 0; \\ -\lambda_1 - \mu_{sp}(1-\beta) - \mu_c, & 0 < z < K_2, \\ -\mu_{sp}(1-\beta) - \mu_c, & z = K_2, \\ 0, & \text{otherwise}, \end{cases}$$

Elements of matrix B_1**:** The sub-matrix B_1 represents the phase distribution of controller, Class CS, and Class ES when the number of packets in the internal buffer is unchanged and there is no packet in the internal buffer (i.e. $n_b(t)$ or w in Eq. (4.1) is equal to 0):

$$B_{1(x,x')} = \begin{cases} B_{11}{}^{(x)}, & x' = x = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$B_{11}{}^{(0)}{}_{(y,y')} = \begin{cases} B_{111}{}^{(y)}, & y' = y = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$B_{111}{}^{(0)}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z + 1, \\ \mu_{sp}(1-\beta), & z' = z - 1, \\ 0, & \text{otherwise}. \end{cases}$$

The diagonal elements of ${B_{111}}^{(0)}_{(z,z')}$ where z is equal to z' is expressed as

$${B_{111}}^{(0)}_{(z,z')} = \begin{cases} -\lambda_1, & z = 0, \\ -\lambda_1 - \mu_{sp}, & 0 < z < K_2, \\ -\mu_{sp}, & z = K_2, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix A_2: The sub-matrix A_2 represents the phase distribution of the controller, Class CS and Class ES when the number of packets in the internal buffer (i.e. $n_b(t)$ or w in Eq. (4.1)) decreases by 1:

$$A_{2(x,x')} = \begin{cases} {A_{21}}^{(x)}, & x' = x, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{21}}^{(x)}_{(y,y')} = \begin{cases} {A_{212}}^{(y)}, & y' = y - 1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{212}}^{(y)}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

4.1.2 Network performance metrics for SPI

The sub-matrices A_0, A_1, B_1, and A_2 are input to MAM which will output π distribution for SPI. This π distribution is used to compute the average packet transfer delay and packet loss probability. These performance metrics are computed using throughputs and the average number of packets in SPI.

Throughput of the internal buffer (denoted by T_{ib}) is given by the sum of probabilities that the internal buffer of the switch has at least one data packet to forward (service rate of μ_{sp}). The throughputs of the internal buffer (T_{ib}) and high priority class (i.e. Class CS) of the switch (T_{cs}) for SPI are the same because a packet in the internal buffer is assumed to be extracted instantaneously after Class CS packet is processed. This assumption is reflected in permissible transitions table for SPI as shown in Table 4.1. Therefore, the throughputs of the internal buffer and Class CS of the switch for SPI is given by:

$$T_{ib} = T_{cs} = \mu_{sp} \sum_{w=1}^{K_3} \sum_{x=0}^{w-1} \sum_{z=0}^{K_2} \pi_{w,x,y,z}. \tag{4.2}$$

Throughput of the Class ES (denoted by T_{es}) for SPI is given by the sum of probabilities that the low priority class of the switch (i.e. Class ES) has at least one data packet to forward (service rate of μ_{sp}) and there is no packet in the high priority class of the switch (i.e. Class CS) in the stationary state, and this is given by:

$$T_{es} = \mu_{sp} \sum_{w=0}^{K_3} \sum_{x=0}^{w} \sum_{z=1}^{K_2} \pi_{w,x,0,z}. \tag{4.3}$$

Throughput of the controller (denoted by T_c) for SPI is given by the sum of probabilities that the controller has at least one control packet to forward (service rate of μ_c) with the condition that there is at least one packet temporarily buffered in the internal buffer of the switch, and this is given by:

$$T_c = \mu_c \sum_{w=1}^{K_3} \sum_{x=1}^{w} \sum_{z=0}^{K_2} \pi_{w,x,y,z}. \tag{4.4}$$

Average number of data packets (denoted by L_{SPI}) is the average number of data packets in SPI where data packets travel only through the switch (the Class ES and the internal buffer) for SPI. Therefore, L_{SPI} is expressed as:

$$L_{SPI} = \sum_{w=0}^{K_3} \sum_{x=0}^{w} \sum_{z=0}^{K_2} (w+z)\pi_{w,x,y,z}. \tag{4.5}$$

Average data packet transfer delay for SPI (denoted by t_{SPI}) is the mean sojourn time of the packet in SPI. The lower value of t_{SPI} results in the lower waiting time of packets in an SDN network with the switch that supports internal buffering. In delay sensitive applications like an industrial automation system, interactive video, and online surgery; a lower t_{SPI} is expected and higher t_{SPI} is unacceptable. The value of t_{SPI} is obtained by applying Little's theorem to Eq. (4.5) which is expressed as:

$$t_{SPI} = L_{SPI}/T_{SPI}, \tag{4.6}$$

where T_{SPI} is the throughput of SPI expressed as:

$$T_{SPI} = T_{ib} + (1-\beta)T_{es}. \tag{4.7}$$

Packet loss probability (denoted by PL_{SPI}) is the average number of packets being blocked or dropped by the Class CS, the Class ES, and the internal buffer out of total incoming packets in SPI. The lower value of PL_{SPI} results in better QoS which is the positive indicator for applications like web-based multimedia and industrial automation.
Assuming independence between the arrival at the Class CS, the Class ES and the internal buffer, the packet loss probability for SPI (PL_{SPI}) is the sum of packet loss probabilities in the Class CS, the Class ES and the internal buffer which is given as:

$$PL_{SPI} = PL_{cs} + PL_{es} + PL_{ib}, \tag{4.8}$$

where PL_{cs}, PL_{es}, and PL_{ib} represents the packet loss probabilities of Class CS, Class ES, and switch's internal buffer, respectively. These packet loss probabilities are expressed as:

$$\begin{aligned} PL_{cs} = PL_{ib} &= 1 - T_{cs}/T_c, \\ PL_{es} &= 1 - T_{es}/\lambda_1. \end{aligned} \tag{4.9}$$

4.1.3 Buffer dimensioning for SPI

For the buffer dimensioning problem in SPI, the switch queue is assumed as M/M/1 (see Section 3.2). The minimum buffer capacities for the Class CS, the Class ES, and the internal buffer of the switch are denoted as K_1, K_2, and K_3, respectively, and can be calculated using Eq. (3.2) as:

$$\begin{aligned} K_1 &\geq \frac{\log[PER]}{\log[\rho_{cs}]}, \\ K_2 &\geq \frac{\log[PER]}{\log[\rho_{es}]}, \\ K_3 &\geq \frac{\log[PER]}{\log[\rho_{ib}]}, \end{aligned} \tag{4.10}$$

where ρ_{cs}, ρ_{es}, and ρ_{ib} are server utilizations at the Class CS, the Class ES, and the internal buffer of the switch, respectively, and are defined as:

$$\rho_{cs} = \frac{\beta\lambda_1}{\mu_{sp}}, \qquad \rho_{es} = \frac{\lambda_1}{\mu_{sp}}, \qquad \rho_{ib} = \frac{\beta\lambda_1}{\mu_{sp}},$$

Therefore, the minimum buffer capacity for the switch in SPI (denoted as K_{SPI}) is the sum of minimum queue capacity for the Class CS, the Class ES and the internal buffer:

$$K_{SPI} = K_1 + K_2 + K_3. \tag{4.11}$$

4.2 Results

In this section, the queueing model for an SDN switch with internal buffering (i.e. SPI) is validated by discrete event simulation. The discrete event simulation based on Monte Carlo simulation for model SPI with pseudo codes is explained in Appendix E. The average packet transfer delay and packet loss probability are the performance metrics of interest. Further, SPI is compared with SPE (i.e. the queueing model for an SDN switch

Table 4.2: Parameter used for analysis and simulation for both SPE and SPI.

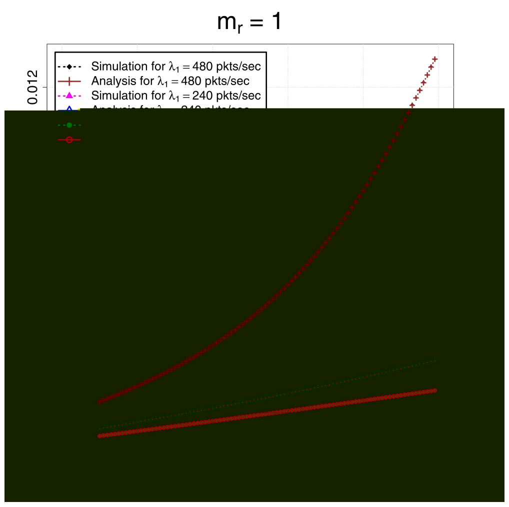
Parameter	**Value**
Table miss probability, β	0.1$\sim$1
Switch processing rate, μ_{sp} (packets/sec)	1000
Arrival rate, λ_1 (packets/sec)	120, 240, 480
Controller to CPU Processing Ratio, m_r	0.1$\sim$2
Bit Error Rate, (BER)	10^{-12}
MTU TCP packet size (byte)	1500

without internal buffering) to identify the benefits and trade-offs of using SPI over SPE.

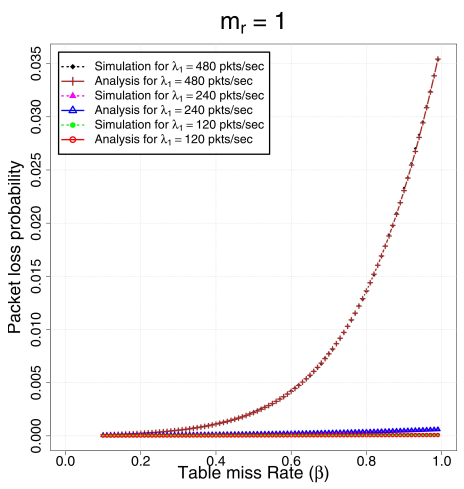
The parameters used for analysis and simulation for both SPE and SPI are shown in Table 4.2. From Table 4.2, the table miss probability β varies from 0.1 to 1 to investigate the performance of the switch in the presence of increasing traffic intensity while the controller to CPU packet processing rate ratio (m_r) varies from 0.1 to 2 to study different processing power disparity that typically exists between the switch and SDN controller.

The CPU processing rate (μ_{sp}) is assumed to be 1000 packets/sec and the external arrival rate (λ_1) is set to $\{120, 240, 480\}$ packets/sec that typify arrivals from a small business premise [118] to a campus area network [66, 120]. To induce packet losses, the queue buffer capacity of Class ES (K_2) is truncated in both SPE and SPI to $\frac{K_2}{2}$ while the values of K_1 and K_3 are determined via buffer dimensioning (Section 3.3.3 and Section 4.1.3) to protect the control packets from being lost or dropped.

The simulations are repeated a hundred times and the 95% confidence intervals (CI) are computed on the basis that the errors are normally distributed.



(a)



(b)

Figure 4.2: Validation of Model SPI in terms of (a) Average packet transfer delay, and (b) Packet loss probability.

4.2.1 Validating queueing model SPI

In this section, the analytical result for SPI is validated by comparing it with discrete event simulation result. Figure 4.2 shows the validation results for SPI for increasing β with $m_r = 1$. The trend of increasing average delay with increasing β from the queueing model track the simulation model very well.

The error percentage between analysis and simulation predictions for both average packet transfer delay and packet loss probability is between 0.6%-2.8% as shown in Fig. 4.2a and Fig. 4.2b.

This range of error is acceptable for analysis as computation of π distributions for non-homogenous QBD processes is known to introduce inaccuracies due to the possibility of a singular matrix becoming nonsingular in machine precision [85].

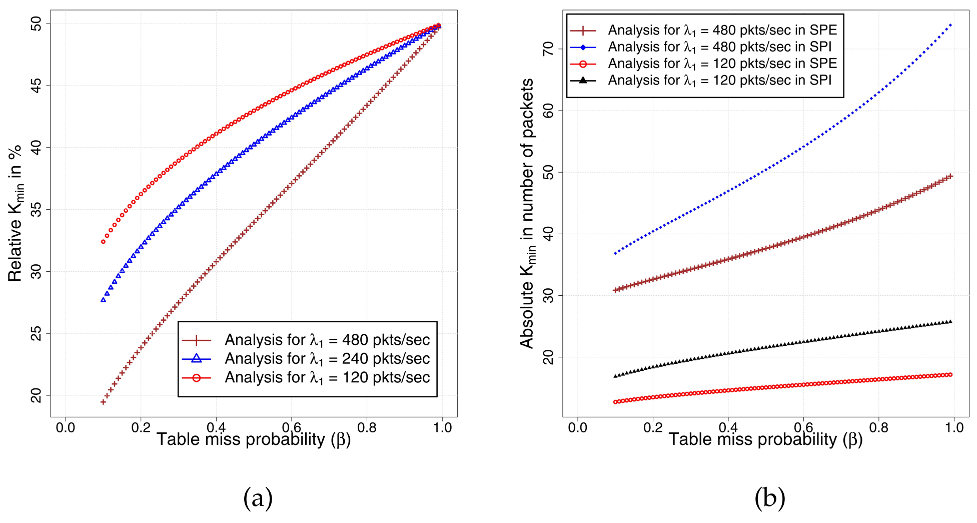


Figure 4.3: Minimum switch queue capacity (K_{min}) between SPE and SPI for $\mu_{sp} = 1000$ pkts/sec and increasing β: (a) Relative difference i.e. ϵ_K and (b) Absolute value.

4.2.2 SPE vs. SPI

4.2.2.1 Relative minimum buffer capacity

In this section, the relative minimum capacity between SPI and SPE denoted as ϵ_K is computed where ϵ_K which is defined as,

$$\epsilon_K = \frac{K_{\text{SPI}} - K_{\text{SPE}}}{K_{\text{SPE}}} \times 100\%.$$

A positive value of ϵ_K means SPI requires more capacity than SPE, while a negative value implies that SPI requires less capacity than SPE.

Figure 4.3a shows the ϵ_K curves and Fig. 4.3b shows the absolute values of K_{SPE} and K_{SPI} for increasing β. From Fig. 4.3a, it can be seen that SPI requires up to 50% more buffer capacity than SPE for increasing table miss probabilities. This is because the switch in SPI requires extra internal buffer to temporarily store packets going to the controller. This result shows the trade-off of using a switch with the internal buffer over a switch without the internal buffer and is expected to be useful for switch designers in balancing costs and meeting quality of service requirements for a switch.

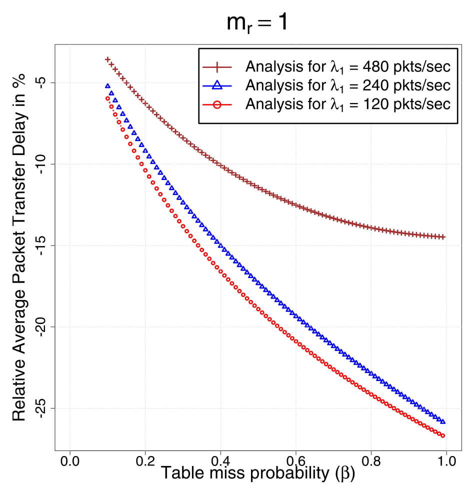
4.2.2.2 Relative average delay

In this section, the average packet transfer delay between SPI (denoted by t_{SPI} given earlier in Eq. 4.6) and SPE (denoted by t_{SPE} given earlier in Eq. 3.30) is compared. In this comparison, the effect of internal buffering in terms of average packet transfer delay is investigated.

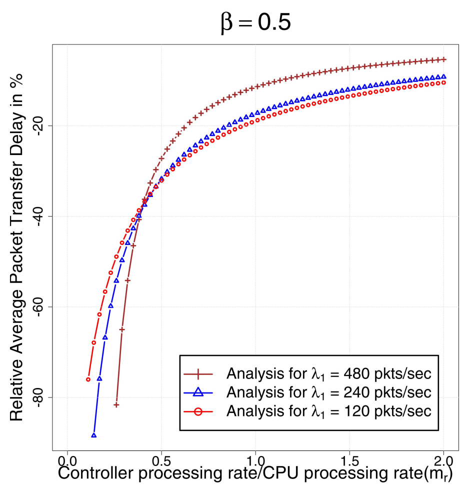
The relative average packet transfer delay (denoted by ϵ_d) between SPE and SPI (both with finite capacity) is calculated as:

$$\epsilon_d = \frac{(t_{SPI} - t_{SPE})}{t_{SPE}} \times 100\%.$$

A negative value of ϵ_d indicates that SPI has lower average delay for packets to travel in the network compared to SPE.



(a)



(b)

Figure 4.4: Relative average packet transfer delay between SPE and SPI i.e. ϵ_t for (a) increasing β and $m_r = 1$; and (b) increasing m_r and $\beta = 0.5$.

Figure 4.4 shows the relative average packet transfer delay between SPI and SPE. Figure 4.4a and Fig. 4.4b show the relative average packet transfer delay between SPI and SPE for increasing β and m_r, respectively.

Varying the table miss probability (β): From Fig. 4.4a, SPI exhibits up to 30% reduction in average delay of the packet compared to SPE. As β increases, more packets are sent to the controller for decisioning, and thus ϵ_d decreases further. Hence, the benefits of having the internal buffer become more significant with increasing β.

The lower delay in SPI is because with the buffering of packets in the switch's internal buffer of SPI, the switch sends a smaller sized message to the controller as compared to SPE. As a result, the packet-in messages in SPI are processed faster than that of SPE. This shows the benefit of utilizing a switch with the internal buffer over a switch without the internal buffer, i.e. reducing the average delay of the packets traversing the SDN switch.

Varying the controller to switch processing ratio (m_r): From Fig. 4.4b, SPI exhibits lower average packet transfer delay with a slower controller (lower m_r) than a faster controller (higher m_r). This is because with a slower controller, the decisioning time increases, which has significant impact in the waiting time for packets in the switch. With internal buffering, packets in a flow awaiting for decisioning from the controller are internally buffered, which allows other packets in a flow to be serviced.

For an SDN network with a faster controller, the decisioning time is also reduced which results in faster flow table updates in the switch. These faster flow table updates shadow the effect of the internal buffer with relatively lower average packet transfer delay.

This shows the benefit of a switch with the internal buffer over a switch without the internal buffer in an SDN network with a slower controller.

4.2.2.3 Relative average packet loss probability

In this section, the packet loss rate between SPE (denoted by PL_{SPE} as in Eq. (3.33)) and SPI (denoted by PL_{SPI} as in Eq. (4.8)) is compared. In this comparison, the effects of packet loss probability in the switch with and without the internal buffer is investigated, computed as

$$\epsilon_{PL} = \frac{(PL_{SPI} - PL_{SPE})}{PL_{SPE}} \times 100\%.$$

A negative value of ϵ_{PL} indicates that SPI has lower packet loss probability delay in the network compared to SPE.

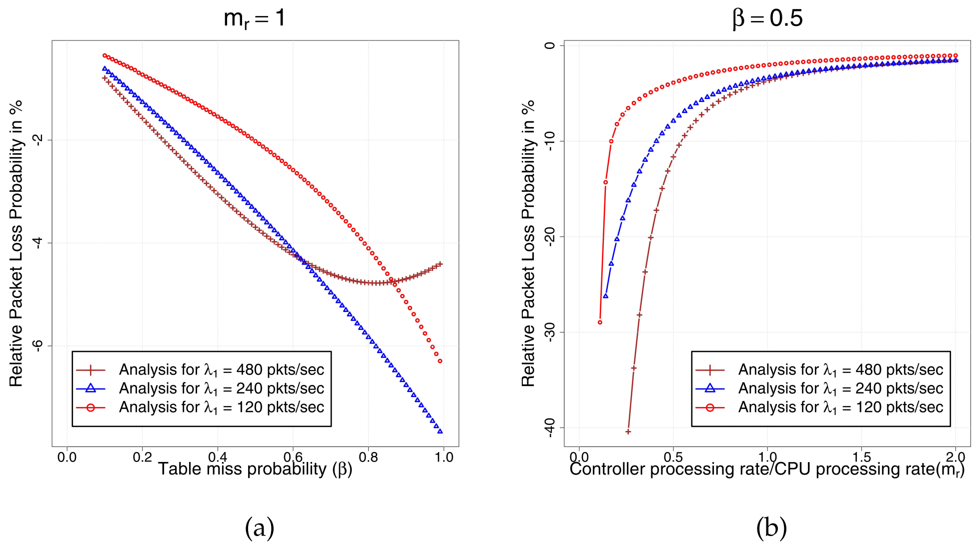


Figure 4.5: Relative average packet loss probability between SPE and SPI i.e. ϵ_t for (a) increasing β and $m_r = 1$; and (b) increasing m_r and $\beta = 0.5$.

Figure 4.5 shows the relative packet loss probability between SPI and SPE. Figure 4.5a and Fig. 4.5b show the relative packet loss probability between SPI and SPE for increasing β and m_r, respectively.

Varying the table miss probability (β): From Fig. 4.5a, SPI exhibits up to 6% reduction in the packet loss probability compared to SPE. This result shows that for increasing β, the internal buffer absorbs the increasing number of data packets that need to be sent to the controller and reduces the loss probability.

The relative loss probability curve for λ_1 = 480 pkts/sec in Fig. 4.5a increases after β reach 0.8. This is because the number of data packets that need to be sent to the controller after β reaches 0.8 is much higher for λ_1 = 480 pkts/sec than λ_1 = 120 or 240 pkts/sec. Due to the limited capacity of the internal buffer, the higher number of data packets for λ_1 = 480 pkts/sec are internally dropped because of which the relative loss probability increases after β reach 0.8.

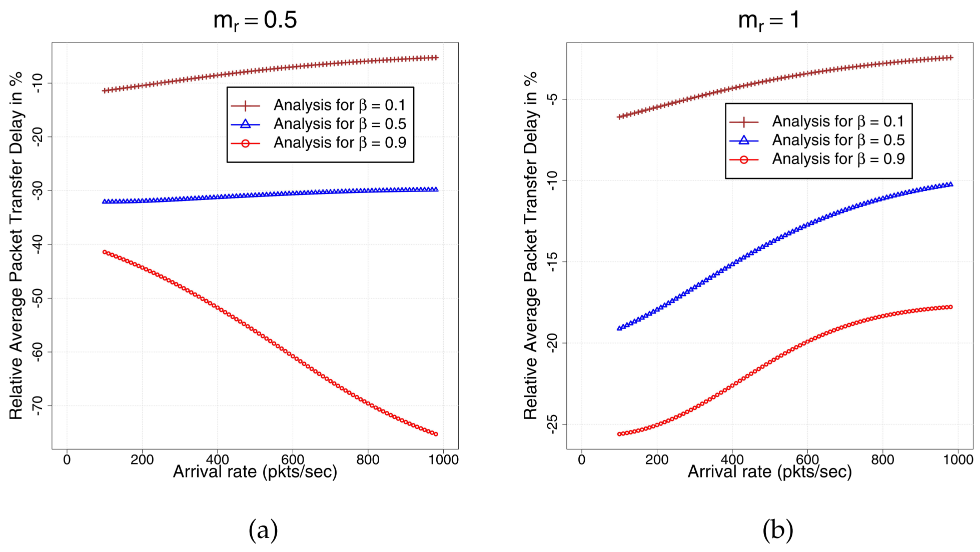
Varying the controller to switch processing ratio **(*m_r*):** From Fig. 4.5b, SPI exhibits up to 50% reduction in the packet loss probability for a slower controller compared to SPE, whereas for a faster controller the reduction is up to 6%.

This result shows that with increasing controller processing power, the waiting time for packets in the switch is reduced. Thus, the output buffer of the switch becomes available for packets arriving at the switch which reduces the blocking of packets. In such a case, the effect of internal buffering in the switch is advantageous with a slower controller than a faster controller.

4.2.3 Increasing packet arrivals (λ_1):

With the fixed switch processing rate, the switch becomes slower with increasing arrival rate. In this section, λ_1 varies from 100 pkts/sec to 990 pkts/sec to study the effect of varying arrival rate in both SPE and SPI. This is done by computing the relative average packet transfer delay and relative packet loss probability for increasing λ_1.

Relative average packet transfer delay: Figure 4.6 shows the relative average packet transfer delay between SPI and SPE for varying λ_1 with varying m_r as seen in Fig. 4.6a ($m_r = 0.5$), Fig. 4.6b ($m_r = 1$), and Fig. 4.6c ($m_r = 2$).



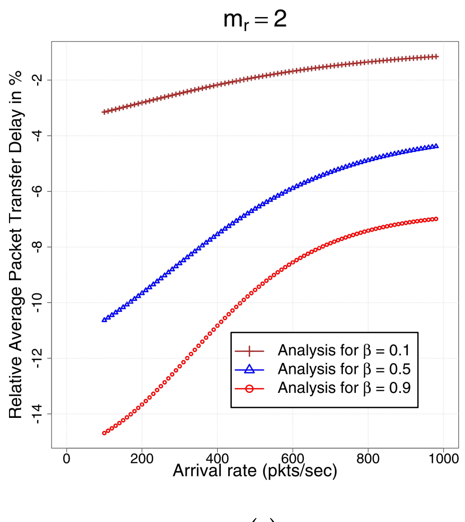


Figure 4.6: Relative average packet transfer delay between SPI and SPE for varying λ_1 with (a) $m_r = 0.5$; (b) $m_r = 1$; and (c) $m_r = 2$.

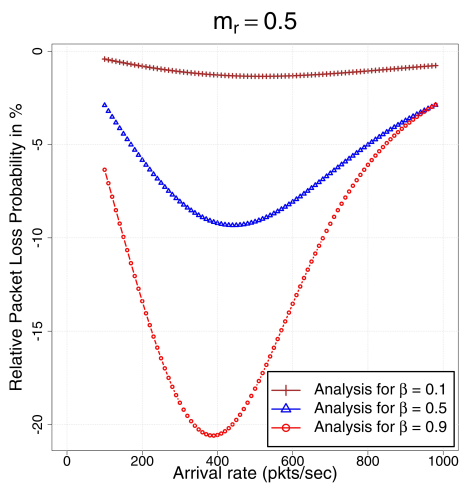
From Fig. 4.6a, the switch with the internal buffer significantly reduces the waiting time of packets up to 70% than the switch without the internal buffer. This is because the waiting time for packets in the switch significantly increases in an SDN network with a slower controller and a slower switch. With temporarily buffering of data packets that require decisioning from a slower controller, a slower switch with the internal buffer provides significance advantage over a slower switch without the internal buffer. However, this benefit comes at the cost of larger memory required for internal buffering.

The curve for β = 0.9 in Fig. 4.6a is different from the other curves because of the amount of traffic that goes to the controller for β = 0.9 increases drastically with the increasing arrival rate. Also, the controller's processing capacity is 0.5 times of the switch's processing capacity (i.e. $m_r = 0.5$). The amount of traffic that goes to the controller for β = 0.9 is much higher than β = 0.1 & 0.5. Therefore, the delay caused by the slower controller for β = 0.9 is also much higher and displays a decreasing trend with the increasing arrival rate.

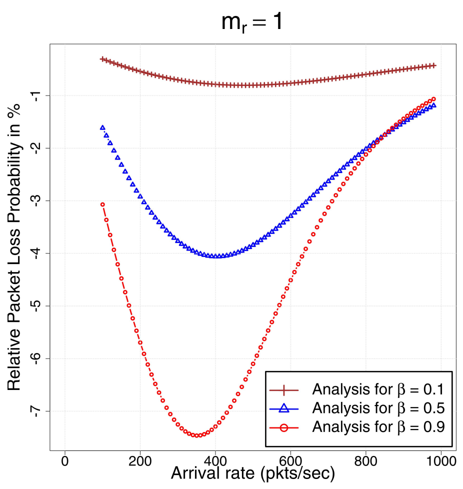
Similarly, for an SDN network with a faster controller and a slower switch, the waiting time of packets awaiting for decisioning from the controller in the switch is reduced. Thus the use of the internal buffer to reduce the overall delay is relatively lower compared to the switch without the internal buffer as seen in Fig. 4.6b and Fig. 4.6c.

Relative packet loss probability: With the increasing λ_1 and β, the number of packets sent to the controller for decisioning is also increased. This results in increase of waiting time for packets and blocking of incoming packets in the switch.

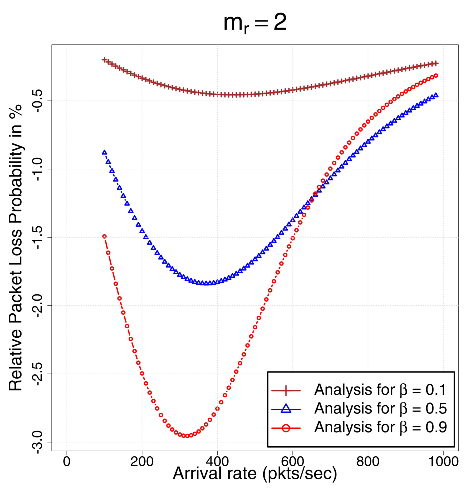
The packets waiting for decisioning are absorbed for a smaller time interval with a faster switch. However, with a slower switch, these absorbed packets are temporarily buffered for longer time interval which results in blocking of packets in the internal buffer as well. This additional packet loss in the internal buffer reduces the benefit of having the internal buffer over the switch without the internal buffer in reduction of packet loss as



(a)



(b)



(c)

Figure 4.7: Relative average packet loss probability between SPE and SPI for varying λ_1 with (a) $m_r = 0.5$; (b) $m_r = 1$; and (c) $m_r = 2$.

seen in Fig. 4.7. In Fig. 4.7, for λ_1 greater than 400 pkts/sec (i.e. slower switch), the relative packet loss probability starts increasing.

4.3 Conclusion

In this chapter, the queueing model for the switch with the internal buffer (SPI) is developed and compared with the switch without the internal buffer (SPE). From this comparison, the following benefits and trade-offs of using the internal buffer over without using the internal buffer are identified:

(a) In an SDN network with a faster controller, the switch with the internal buffer reduces the average delay by 30% and packet loss probability by 7% of the packets in an SDN. These numbers increase with a slower controller where the switch with the internal buffer reduces the average delay by 85% and packet loss probability by 40% of the packets in an SDN.

(b) The trade-off is that up to 50% extra switch buffer capacity is required.

4.4 Summary

Based on the investigation from this chapter, the developed queueing model for an SDN switch with the internal buffer showed better performance than the switch without the internal buffer. The switch with the internal buffer gave a lower delay and packet loss than the switch without the internal buffer at the cost of higher memory required for internal buffering. However in a computer network, this trade-off is acceptable for delay and loss sensitive applications where QoS is of the utmost importance.

In the following chapter, hardware-based SDN switches with and without the internal buffer are modelled and analysed.

Chapter 5

Hardware Switching in SDN Switches

Prior to this chapter, a software-based SDN switch is studied and investigated. In this chapter, the effect of hardware switching in SDN switches is studied. The block diagram of a hardware-based SDN switch where the switch maintains flow tables in both hardware and software is shown in Fig. 5.1. The hardware and software flow tables are synchronized through a middleware layer on the switch to avoid duplicate entries and to ensure consistent forwarding behavior [123, 124].

There are four important phases that an SDN model with a hardware switch must capture. Phase (1), the first packet of a flow arrives at the specialised hardware in the switch that maintain hardware flow table entries (FTE) and there is no matching FTE for the packet. Phase (2), a packet with the matching FTE in the TCAM is serviced by the ASIC and forwarded to the destination, otherwise a packet without a matching FTE in TCAM is matched against the FTE in SDRAM and processed by the CPU for forwarding to the destination. In phase (3), a packet without any matching entry in TCAM or SDRAM is forwarded to the controller. In phase (4), the controller feeds the forwarding information back to the switch and updates the flow tables in both TCAM and SDRAM. Finally, the packet is

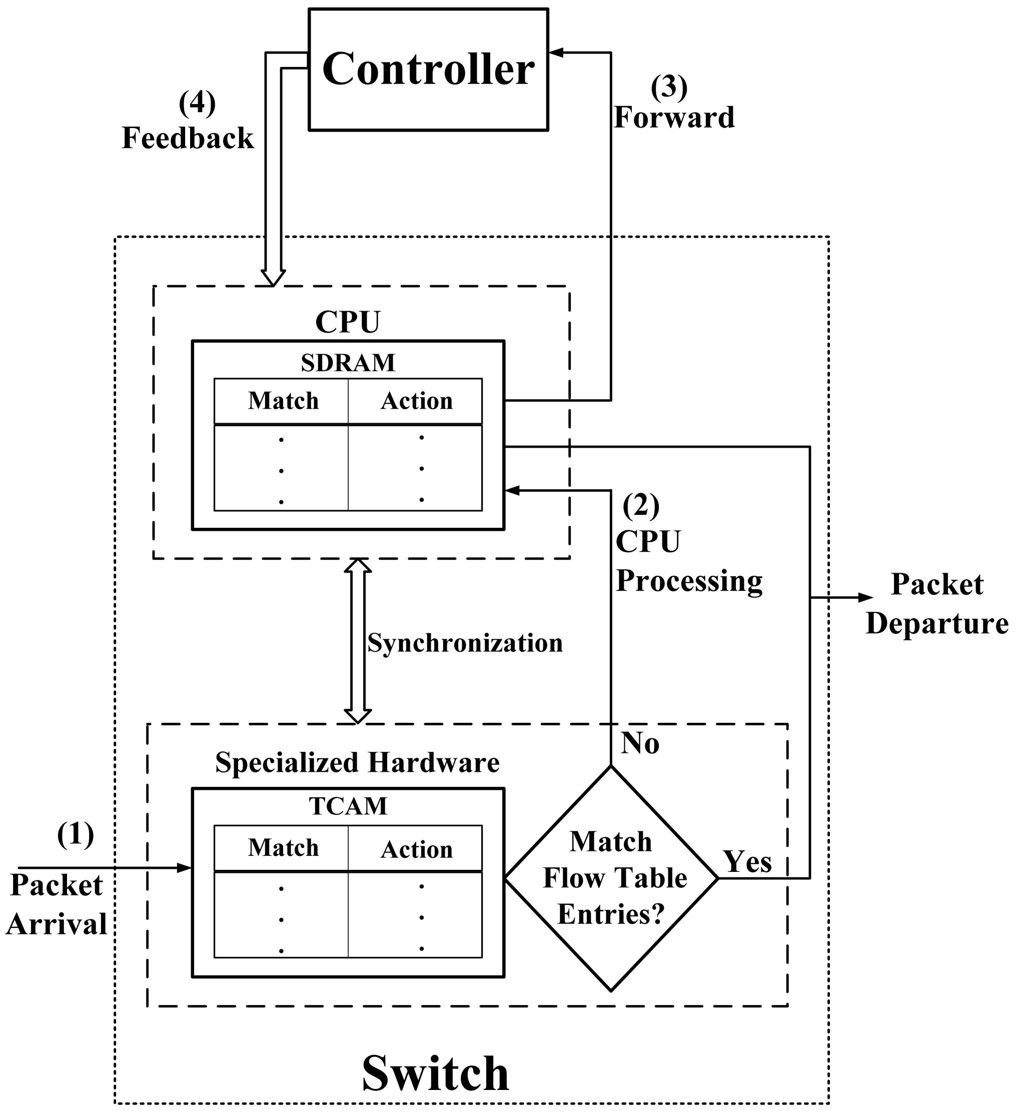


Figure 5.1: Generic model for a hardware-based SDN switch with the specialised hardware and the CPU.

serviced by the CPU and forwarded to the destination.

Based on these four important phases, queueing models for a hardware-based SDN switch with and without the internal buffer are developed in the following sections.

5.1 Model HPE: a hardware switch without the internal buffer

Figure 5.2 shows the queueing model for a hardware-based SDN switch without the internal buffer which is named Model HPE. Similar to a software switch without the internal buffer in Fig. 3.2 (i.e Model SPE), "E" and "P" in HPE refers to the full encapsulation of a data packet and the use of priority queues in the CPU respectively. The "H" in HPE refers to the hardware switch. The priority queues in the hardware switch provide isolation between the packets arriving from the controller and packets to be processed by CPU when there is no matching FTE in the hardware table stored in the switch's TCAM. In this model, non-preemptive priority queues are used for the CPU similar to that in Model SPE for the software switch.

Class HP represents the low priority class of CPU for an external data packet that has no matching entry in the hardware table maintained by the switch's specialised hardware. This packet is sent to the controller by the switch's specialised hardware with the probability β. Class CP represents the high priority class for packets fed back to the CPU from the controller and must be forwarded out to the destination. It is assumed that the CPU synchronises the flow tables with specialised hardware as shown in Fig. 5.1. Both Class CP and Class HP queues share service rate μ_{sp} while the switch's hardware queue has a service rate of μ_{sh}.

The packet processing in HPE can be explained in five steps as shown in Fig. 5.2: (1) external data packets arrives at the specialised hardware

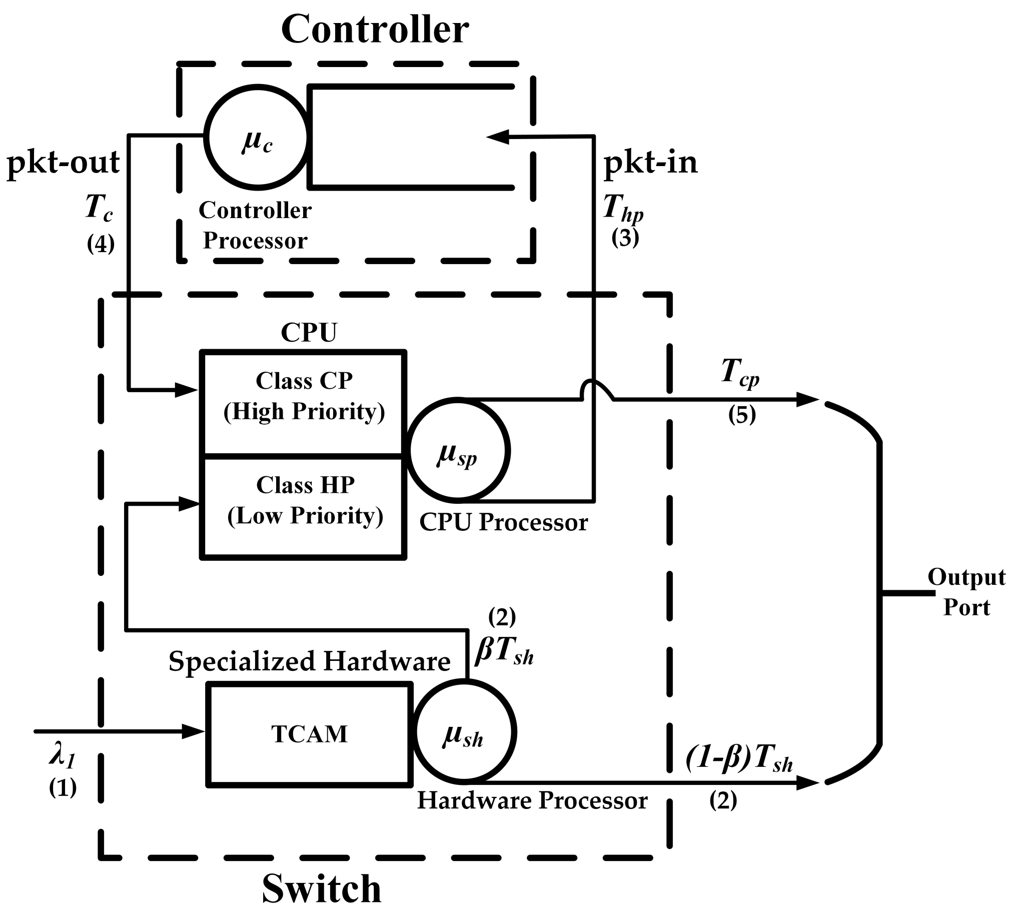


Figure 5.2: Model HPE – hardware switch modelled with two servers to reflect the presence of network processing functions.

of the switch, (2) data packets are forwarded to Class HP of CPU if the specialised hardware in the switch does not have a matching FTE **or** forwarded to destination through output port, (3) data packets are forwarded to the controller encapsulated with packet-in control message, (4) controller feeds back the forwarding information with packet-out message to Class CP of the CPU, (5) finally the CPU processes the control packets in Class CP, updates and synchronises the flow table with specialised hardware, and forwards data packets to the destination through an output port.

The HPE is modelled as a continuous time Markov process with four state variables, $\{(n_c(t), n_{cp}(t), n_{hp}(t), n_{sh}(t)), t \geq 0\}$. The state variables denoted by $n_c(t)$, $n_{cp}(t)$, $n_{hp}(t)$, and $n_{sh}(t)$ represent the number of packets in controller, Class CP, Class HP, and switch hardware, respectively. Let the Markov process at time t be defined as:

$$\{n_c(t), n_{cp}(t), n_{hp}(t), n_{sh}(t)\} = \{w, x, y, z\}. \tag{5.1}$$

where $w \in \mathbb{Z}_{+}$, $x \in \mathbb{Z}_{+}^{\leq K_1}$, $y \in \mathbb{Z}_{+}^{\leq K_2}$, and $z \in \mathbb{Z}_{+}^{\leq K3}$. The queue capacity for the CPU of the switch is equal to K. It is assumed that Class CP and Class HP have a queue capacity of K_1 and K_2 respectively with the total queue capacity of the CPU as K. Similarly, the queue capacity for switch hardware is equal to K_3.

The permissible transitions for the Markov process $\{(n_c, n_{cp}, n_{hp}, n_{sh})\}$ are listed in Table 5.1. With the help of permissible transitions in Table 5.1, the sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the transition rate generator matrix determine (Q) for Model HPE are derived. These sub-matrices are further used to compute the stationary distribution probability (π) for HPE.

5.1.1 Generator matrix

Here, sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the generator matrix for HPE are derived.

Table 5.1: Permissible transitions for Model HPE.

Event	From	To	Rate
One packet arrives at the switch hardware.	(w, x, y, z)	$(w, x, y, z+1)$	λ_1
One packet departs from hardware to out of the system.	$(w, x, y, z>0)$	$(w, x, y, z-1)$	$\mu_{sh}(1-\beta)$
One packet arrives at Class HP for CPU processing.	$(w, x, y, z>0)$	$(w, x, y+1, z-1)$	$\mu_{sh}\beta$
One packet forwarded from Class HP to controller.	$(w, 0, y>0, z)$	$(w, 0, y-1, z)$	μ_{sp}
One packet serviced by Controller to Class CP.	(w, x, y, z)	$(w, x+1, y, z)$	μ_c
One packet processed by CPU to out of the system.	$(w, x>0, y, z)$	$(w, x-1, y, z)$	μ_{sp}

Elements of matrix A_0: The sub-matrix A_0 represents the phase distribution of Class CP, Class HP, and switch hardware when the number of packets in the controller (i.e. $n_c(t)$ or w in Eq. 5.1) increases by 1.

$$A_{0(x,x')} = \begin{cases} {A_{01}}^{(x)}, & x' = x = 0, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$ {A_{01}}^{(0)}{}_{(y,y')} = \begin{cases} {A_{012}}^{(y)}, & y' = y - 1, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$ {A_{012}}^{(1\,\le\, y\,\le\, K_2\,+\,1)}{}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z - 1, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix A_1: The sub-matrix A_1 represents the phase distribution of Class CP, Class HP, and switch hardware when the number of packets in the controller remains unchanged and there are some packets in the controller (i.e. $n_c(t)$ or w in Eq. 5.1 is a positive integer that remains unchanged).

$$A_{1(x,x')} = \begin{cases} {A_{11}}^{(x)}, & x' = x, \\ {A_{12}}^{(x)}, & x' = x - 1, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$ {A_{11}}^{(0\,\le\, x\,\le\, K_1\,+\,1)}{}_{(y,y')} = \begin{cases} {A_{110}}^{(y)}, & y' = y + 1, \\ {A_{111}}^{(y)}, & y' = y, x = 0, \\ {\tilde{A}_{111}}^{(y)}, & y' = y, x \neq 0, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$A_{12}{}^{(1\,\leq\,x\,\leq\,K_1\,+\,1)}{}_{(y,y')} = \begin{cases} A_{121}{}^{(y)}, & y' = y, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$A_{111}{}^{(0\,\leq\,y\,\leq\,K_2\,+\,1)}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z+1, \\ \mu_{sh}(1-\beta), & y = 0, z' = z-1, \\ 0, & \text{otherwise,} \end{cases}$$

$\tilde{A}_{111}{}^{(0\,\leq\,y\,\leq\,K_2\,+\,1)} = A_{111}{}^{(0\,\leq\,y\,\leq\,K_2\,+\,1)}$,

$$A_{110}{}^{(0\,\leq\,y\,<\,K_2\,+\,1)}{}_{(z,z')} = \begin{cases} \mu_{sh}\beta, & z' = z, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$A_{121}{}^{(0\,\leq\,y\,\leq\,K_2\,+\,1)}{}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

The diagonal elements of A_{111} and $\tilde{A}_{111}$ are given as,

$$A_{111}{}^{(y)}{}_{(z,z)} = \begin{cases} -\lambda_1 - \mu_c, & y = 0, z = 0, \\ -\lambda_1 - \mu_c - \mu_{sp}, & 0 < y \leq K_2 + 1, z = 0, \\ -\lambda_1 - \mu_c - \mu_{sh}, & y = 0, 0 < z \leq K_3, \\ -\lambda_1 - \mu_c - \mu_{sh} - \mu_{sp}, & 0 < y \leq K_2 + 1 \text{ and,} \\ & 0 < z \leq K_3 \\ -\mu_c - \mu_{sh}, & y = 0, z = K_3 + 1, \\ -\mu_c - \mu_{sh} - \mu_{sp}, & 0 < y \leq K_2 + 1 \text{ and,} \\ & z = K_3 + 1, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$\tilde{A_{111}}^{(y)}{}_{(z,z)} = \begin{cases} A_{111}{}^{(y)} - \mu_{sp} I_e, & y = 0, \\ A_{111}{}^{(y)}, & y \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

where, I_e is the identity matrix.

Elements of matrix A_2**:** The sub-matrix A_2 represents the phase distribution of Class CP, Class HP, and switch hardware when the number of packets in the controller (i.e. $n_c(t)$ or w in Eq. 5.1) decreases by 1.

$$A_{2(x,x')} = \begin{cases} A_{20}{}^{(\mathrm{x})}, & x' = x + 1, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$A_{20}{}^{(0 \leq x \leq K_1 + 1)}{}_{(y,y')} = \begin{cases} A_{201}{}^{(\mathrm{y})}, & y' = y, \\ 0, & \text{otherwise.} \end{cases}$$

where,

$$A_{201}{}^{(0 \leq y \leq K_2 + 1)}{}_{(z,z')} = \begin{cases} \mu_c, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix B_1**:** The sub-matrix B_1 is identical to A_1 that represents the phase distributions of Class CP, Class HP and switch hardware when the number of packets in controller (i.e. $n_c(t)$ or w in Eq. 5.1 is equal to 0).

$$\therefore B_1 = A_1(\text{ for } \mu_c = 0).$$

5.1.2 Network performance metrics for HPE

With these sub-block matrices and matrix geometric solution, we can compute the stationary distribution probabilities $(\pi_{w,x,y,z})$ for w packets in the controller, x packets in Class CP, y packets in Class HP, and z packets in the switch hardware. The performance metrics like throughput of the controller and switch, the average packet transfer delay in the switch, and average packet loss in the switch can be computed using $(\pi_{w,x,y,z})$.

Throughput of Class CP (denoted by T_{cp}) for Model HPE is given by the sum of probabilities that the Class CP of the CPU has at least one control packet from the controller to process with service rate of μ_{sp} and this is given by:

$$T_{cp} = \mu_{sp} \sum_{w=0}^{\infty} \sum_{x=1}^{K_1} \sum_{y=0}^{K_2} \sum_{z=0}^{K_3} \pi_{w,x,y,z}. \tag{5.2}$$

Throughput of Class HP (denoted by T_{hp}) for Model HPE is given by the sum of probabilities that the Class HP of the CPU has at least one data packet to forward to the controller with service rate of μ_{sp} and no control packet in the Class CP in the stationary state, and this is given by:

$$T_{hp} = \mu_{sp} \sum_{w=0}^{\infty} \sum_{y=1}^{K_2} \sum_{z=0}^{K_3} \pi_{w,0,y,z}. \tag{5.3}$$

Throughput of specialised hardware queue (denoted by T_{hp}) for Model HPE is given by the sum of probabilities that the specialised hardware switch has at least one data packet to forward with service rate of μ_{sh} and this is given by:

$$T_{sh} = \mu_{sh} \sum_{w=0}^{\infty} \sum_{x=0}^{K_1} \sum_{y=0}^{K_2} \sum_{z=1}^{K_3} \pi_{w,x,y,z}. \tag{5.4}$$

Throughput of Controller (denoted by T_c) for Model HPE is given by the sum of probabilities that the controller has at least one control packet to forward to Class CP with service rate of μ_c and this is given by:

$$T_c = \mu_c \sum_{w=1}^{\infty} \sum_{x=0}^{K_1} \sum_{y=0}^{K_2} \sum_{z=0}^{K_3} \pi_{w,x,y,z}. \tag{5.5}$$

Average number of data packets (denoted by L_{HPE}) is the average number of data packets in HPE where data packets travel through the switch (the CPU and the specialised hardware) and the controller. Therefore, L_{HPE} is expressed as:

$$L_{HPE} = \sum_{w=0}^{\infty} \sum_{x=0}^{K_1} \sum_{y=0}^{K_2} \sum_{z=0}^{K_3} (w + x + y + z)\pi_{w,x,y,z}. \tag{5.6}$$

Average data packet transfer delay for HPE (denoted by t_{HPE}) is the mean sojourn time of a data packet in an SDN network with a single controller and hardware switch without the internal buffer. Applying Little's formula on Eq .(5.6), the average time to traverse packet in Model HPE is derived and given as,

$$t_{HPE} = \frac{L_{HPE}}{T_{HPE}}, \tag{5.7}$$

where T_{HPE} is the total throughput of Model HPE and given as,

$$T_{HPE} = T_{cp} + (1 - \beta)T_{sh}. \tag{5.8}$$

Packet loss probability (denoted by PL_{HPE}) is the total average packet loss probability of the Class CP (PL_{cp}), the Class HP (PL_{hp}), and the specialised hardware (PL_{sh}) for Model HPE. Assuming independence of packet arrivals between Class CP, Class HP, and the specialised hardware queue; PL_{cp}, PL_{hp} and PL_{sh} represent the average number of packets being blocked or dropped by Class CP, Class HP and switch hardware out of the total incoming packets in the respective queue. The packet loss probabilities PL_{cp}, PL_{hp}, and PL_{sh} for Model HPE are expressed as,

$$\begin{aligned} PL_{cp} &= 1 - T_{cp}/T_c, \\ PL_{hp} &= 1 - T_{hp}/T_{sh}, \\ PL_{sh} &= 1 - T_{sh}/\lambda_1. \end{aligned} \tag{5.9}$$

The average packet loss probability for Model HPE is given as,

$$PL_{HPE} = 1 - T_{HPE}/\lambda_1. \tag{5.10}$$

The validity of the expressions given for HPE are contingent on the stability condition which can be derived using the traffic equilibrium equation [125]. Let Tr_c, Tr_{cp}, Tr_{hp} and Tr_{sh} denote the traffic intensities at the controller, Class CP, Class ES of the CPU and switch hardware respectively. The queueing network represented by HPE is stable if $(Tr_c - \mu_c) < 0$. Using the traffic equilibrium equations, we get,

$$Tr_c = Tr_{hp}, \quad Tr_{cp} = Tr_c, \quad Tr_{hp} = \beta Tr_{sh}, \quad Tr_{sh} = \lambda_1.$$

Solving the traffic equations for HPE and using the condition of $(Tr_c - \mu_c) < 0$, we get the stability condition from [11] for HPE which is the same as that of SPE, i.e.,

$$\beta\lambda_1 - \mu_c < 0. \tag{5.11}$$

5.2 Model HPI: a hardware switch with the internal buffer

Similar to "SPI" for the software switch with the internal buffer, the queueing model for a hardware switch with the internal buffer is named Model HPI, where "I" refers to internal buffering. HPI is an extension of SPI, with one additional server and queue for specialised hardware with M/M/1/K distribution.

As shown in Fig. 5.3, the switch has two servers, one for specialised hardware (referred to as hardware processor and denoted by μ_{sh}) and the other one for the CPU (referred to as CPU processor and denoted by μ_{sp}). Similar to SPI, the CPU is modelled as finite capacity with non-preemptive two-priority class queues; Class HP (similar to Class ES for SPI) as a low priority, Class CP (similar to Class CS for SPI) as a high priority.

The packet processing in HPI can be explained in five steps as shown in Fig. 5.3: (1) external data packets arrive at specialised hardware of the switch, (2) data packets are forwarded to Class HP of CPU if the spe-

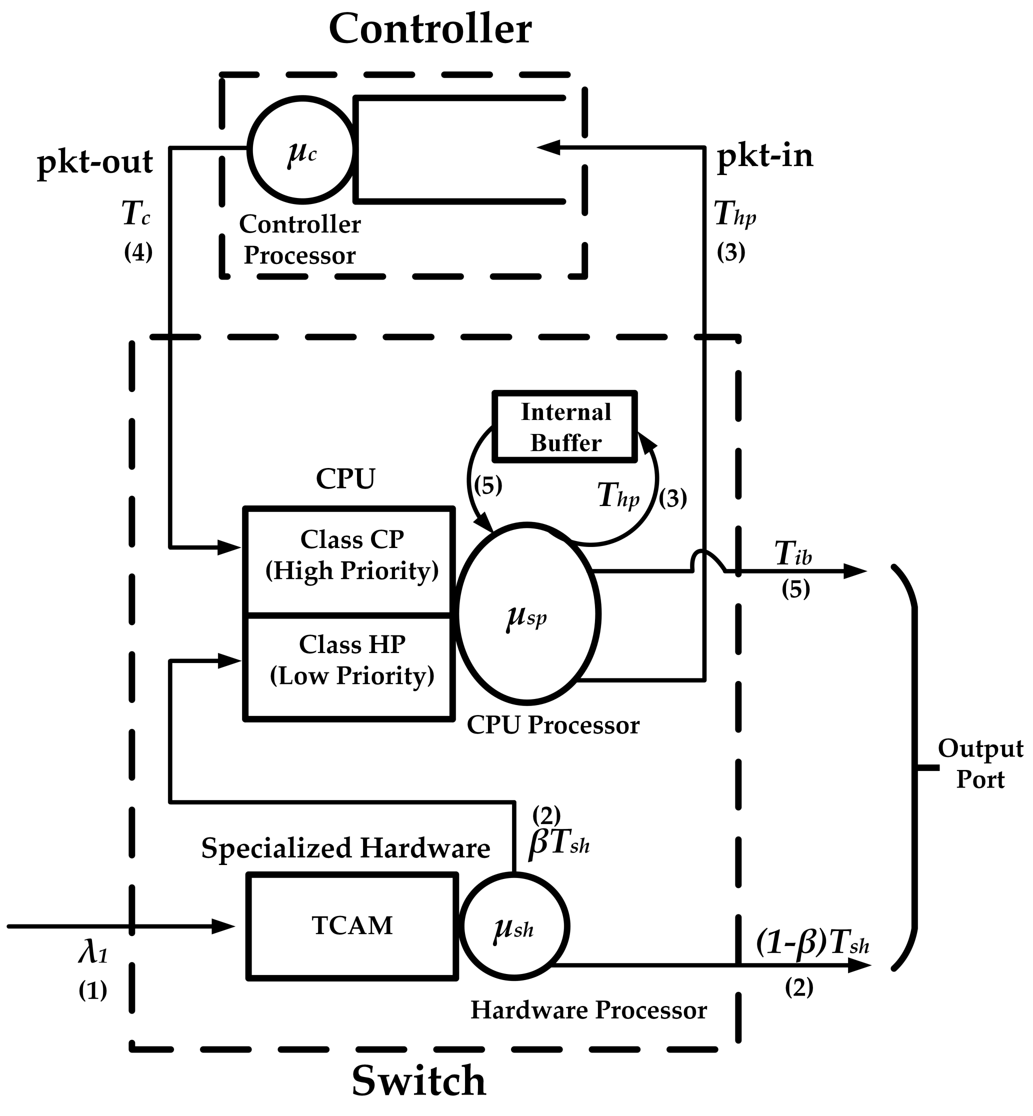


Figure 5.3: Model HPI – hardware switch modelled with two servers and internal buffer to realise the internal buffering.

cialised hardware in the switch does not have a matching FTE **or** forwarded to destination through output port, (3) data packets are temporarily buffered in the internal memory and a fraction of the data packet is forwarded to the controller encapsulated with packet-in control message, (4) controller feedback the forwarding information with packet-out message to Class CP of the CPU, (5) finally the CPU processes the control packets in Class CP, updates and synchronises the flow table with specialised hardware, extracts temporarily buffered data packet from the internal buffer and forwards to the destination through an output port.

HPI is modelled as a continuous time Markov process with five state variables, $\{(n_b(t), n_c(t), n_{cp}(t), n_{hp}(t), n_{sh}(t)), t \geq 0\}$. The state variables denoted by $n_b(t)$, $n_c(t)$, $n_{cp}(t)$, $n_{hp}(t)$, and $n_{sh}(t)$ represent the number of packets in the internal buffer, controller, Class CP, Class HP, and specialised hardware respectively.

Similar to SPI, queue capacities of the internal buffer, Class CP, and Class HP are $K_3, K_1,$ and K_2 respectively; and the controller is assumed to have infinite capacity. The queue capacity of specialised hardware is K_4. Let the Markov process at time t be defined as:

$$\{n_b(t), n_c(t), n_{cp}(t), n_{hp}(t), n_{sh}(t)\} = \{v, w, x, y, z\} \tag{5.12}$$

where $v \in \mathbb{Z}_+^{\leq K_3}$, $w \in \mathbb{Z}_+$, $x \in \mathbb{Z}_+^{\leq K_1}$, $y \in \mathbb{Z}_+^{\leq K_2}$, and $z \in \mathbb{Z}_+^{\leq K_4}$. The number of packets in the controller and the Class CP are dependent on the number of temporarily buffered packets in the internal buffer. Therefore, the state space of the controller can be rewritten as $w \in \mathbb{Z}_+^{\leq v}$ and and the state space of the Class CP is limited by the number of packets in the internal buffer and the contoller i.e., $x = v - w$ as discussed in Section 4.1.

Due to the dependency of $n_c(t)$ and $n_{cp}(t)$ on the internal buffer, the process governing the number of packets in HPI is also a nonhomogenous QBD process with the internal buffer as a level variable; controller, Class CP, Class HP, and specialised hardware as phase variables. The permissible transitions for the Markov chain $\{(n_b(t), n_c(t), n_{cp}(t), n_{hp}(t), n_{sh}(t))\}$

Table 5.2: Permissible Transitions for Model HPI.

Event	**From**	**To**	**Rate**
One packet arrives at switch hardware.	(v, w, x, y, z)	$(v, w, x, y, z + 1)$	λ_1
One packet departs from hardware to out of the system (HPI).	$(v, w, x, y, z > 0)$	$(v, w, x, y, z - 1)$	$\mu_{sh}(1 - \beta)$
One packet arrives at Class HP for CPU processing.	$(v, w, x, y, z > 0)$	$(v, w, x, y + 1, z - 1)$	$\mu_{sh}\beta$
One packet departs from Class HP to the internal buffer and subsequently one packet-in message is sent to the controller.	$(v, w, 0, y > 0, z)$	$(v + 1, w + 1, 0, y - 1, z)$	μ_{sp}
One packet serviced by the controller to Class CP.	$(v > 0, w > 0, x, y, z)$	$(v > 0, w - 1, x + 1, y, z)$	μ_c
One packet_out in Class CP is processed and subsequently one packet departs from the internal buffer to out of the system (HPI).	$(v > 0, w, x > 0, y, z)$	$(v - 1, w, x - 1, y, z)$	μ_{sp}

are listed in Table 5.2. These transitions help in deriving sub-matrices (A_0, A_1, B_1, and A_2) of the generator matrix (Q) for HPI. These sub-matrices are input to matrix geometric solution to compute the stationary distribution probability (π) which is used to determine performance metrics for HPI.

5.2.1 Generator matrix

Here, sub-matrices (denoted by A_0, A_1, B_1, and A_2) of the generator matrix for HPI are derived.

Elements of matrix A_0***:*** The sub-matrix A_0 represents the phase distribution of controller, Class CP, Class HP, and specialised hardware when the number of packets in the internal buffer (i.e. $n_b(t)$ or v in Eq. (5.12)) increases by 1:

$$A_{0(w,w')} = \begin{cases} {A_{00}}^{(w)}, & w' = w + 1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{00}}^{(w)}{}_{(x,x')} = \begin{cases} {A_{001}}^{(x)}, & x' = x = 0, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{001}}^{(0)}{}_{(y,y')} = \begin{cases} {A_{0012}}^{(y)}, & y' = y - 1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{0012}}^{(y)}{}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix A_1***:*** The sub-matrix A_1 represents the phase distribution of controller, Class CP, Class HP, and specialised hardware when the number of packets in the internal buffer remain unchanged and there are some packets in the internal buffer (i.e. $n_b(t)$ or v in Eq. (5.12) is a positive integer that remain unchanged):

$$A_{1(w,w')} = \begin{cases} {A_{11}}^{(w)}, & w' = w, \\ {A_{12}}^{(w)}, & w' = w-1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{11}}^{(w)}{}_{(x,x')} = \begin{cases} {A_{111}}^{(x)}, & x' = x, \\ 0, & \text{otherwise,} \end{cases}$$

and

$${A_{12}}^{(w)}{}_{(x,x')} = \begin{cases} {A_{120}}^{(x)}, & x' = x+1, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{111}}^{(x)}{}_{(y,y')} = \begin{cases} {A_{1111}}^{(y)}, & y' = y, \\ {A_{1110}}^{(y)}, & y' = y+1, \\ 0, & \text{otherwise,} \end{cases}$$

and

$${A_{120}}^{(x)}{}_{(y,y')} = \begin{cases} {A_{1201}}^{(y)}, & y' = y, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$${A_{1111}}^{(y)}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z+1, \\ \mu_{sh}(1-\beta), & z' = z-1, \\ 0, & \text{otherwise,} \end{cases}$$

$${A_{1110}}^{(y)}{}_{(z,z')} = \begin{cases} \mu_{sh}\beta, & z' = z-1, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$A_{1201}{}^{(y)}{}_{(z,z')} = \begin{cases} \mu_c, & z' = z, \\ 0, & \text{otherwise.} \end{cases}$$

The diagonal elements of $A_{1111}{}^{(y)}{}_{(z,z')}$ where z is equal to z' has the four distinct cases:

(i) when there is no packet in the controller (i.e. $n_c(t)$ or w in Eq. (5.12) is equal to 0),

$$A_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_{sp}, & 0 \leq y \leq K_2, \\ & z = 0; \\ -\lambda_1 - \mu_{sh} - \mu_{sp}, & 0 \leq y < K_2, \\ & 0 < z < K_4; \\ -\lambda_1 - \mu_{sh}(1-\beta) - \mu_{sp}, & y = K_2, \\ & 0 < z < K_4; \\ -\mu_{sh} - \mu_{sp}, & 0 \leq y < K_2, \\ & z = K_4; \\ -\mu_{sh}(1-\beta) - \mu_{sp}, & y = K_2, \\ & z = K_4; \\ 0, & \text{otherwise,} \end{cases}$$

(ii) when the number of packets in the controller is less than the internal buffer which is not full i.e. $0 < w < v$ and $v < K_3$,

$$A_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_{sp} - \mu_c, & 0 \le y \le K_2, \\ & z = 0; \\ -\lambda_1 - \mu_{sp} - \mu_{sh} - & 0 \le y < K_2, \\ \mu_c, & 0 < z < K_4; \\ -\mu_{sp} - \mu_{sh} - \mu_c, & 0 \le y < K_2, \\ & z = K_4; \\ -\lambda_1 - \mu_{sp} & y = K_2, \\ -\mu_{sh}(1-\beta) - \mu_c, & 0 < z < K_4; \\ -\mu_{sp} - \mu_{sh}(1-\beta) & y = K_2, \\ - \mu_c, & z = K_4; \\ 0, & \text{otherwise}, \end{cases}$$

(iii) when the number of packets in the controller is equal to that in the internal buffer which is not full i.e. $w = v$ and $v < K_3$,

$$A_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_c, & y = z = 0; \\ -\lambda_1 - \mu_{sh} - \mu_c, & y = 0, \\ & 0 < z < K_4; \\ -\mu_{sh} - \mu_c, & y = 0, \\ & z = K_4; \\ -\lambda_1 - \mu_{sp} - \mu_c, & 0 < y \le K_2, \\ & z = 0; \\ -\lambda_1 - \mu_{sp} - \mu_{sh} - & 0 < y \le K_2, \\ \mu_c, & 0 < z < K_4; \\ -\mu_{sp} - \mu_{sh} - \mu_c, & 0 < y \le K_2, \\ & z = K_4; \\ -\lambda_1 - \mu_{sp} - \mu_c & y = K_2, \\ - \mu_{sh}(1 - \beta), & 0 < z < K_4; \\ -\mu_{sp} - \mu_{sh}(1-\beta) & y = K_2, \\ - \mu_c, & z = K_4; \\ 0, & \text{otherwise}, \end{cases}$$

(iv) when the number of packets in the controller and the internal buffer are equal to the queue size of the internal buffer i.e. $w = v = K_3$,

$$A_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1 - \mu_c, & 0 \le y \le K_2, \\ & z = 0; \\ -\lambda_1 - \mu_{sh} - \mu_c, & 0 \le y < K_2, \\ & 0 < z < K_4; \\ -\mu_{sh} - \mu_c, & 0 \le y < K_2, \\ & z = K_4; \\ -\lambda_1 - \mu_{sh}(1-\beta) & y = K_2, \\ -\ \mu_c, & 0 < z < K_4; \\ -\mu_{sh}(1-\beta) - \mu_c, & y = K_2, \\ & z = K_4; \\ 0, & \text{otherwise}, \end{cases}$$

Elements of matrix B_1***:*** The sub-matrix B_1 represents the phase distribution of the controller, Class CP, Class HP, and specialised hardware when the number of packets in the internal buffer remain unchanged and there is no packet in the internal buffer (i.e. $n_b(t)$ or v in Eq. (5.12) is equal to 0:)

$$B_{1(w,w')} = \begin{cases} B_{11}{}^{(w)}, & w' = w = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$B_{11}{}^{(0)}{}_{(x,x')} = \begin{cases} B_{111}{}^{(x)}, & x' = x = 0, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$B_{111}{}^{(0)}{}_{(y,y')} = \begin{cases} B_{1111}{}^{(x)}, & y' = y, \\ 0, & \text{otherwise}. \end{cases}$$

where,

$$B_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} \lambda_1, & z' = z+1, \\ \mu_{sh}(1-\beta), & z' = z-1, \\ 0, & \text{otherwise.} \end{cases}$$

The diagonal elements of $B_{1111}{}^{(y)}{}_{(z,z')}$ where z is equal to z' are expressed as

$$B_{1111}{}^{(y)}{}_{(z,z')} = \begin{cases} -\lambda_1, & y=0, z=0; \\ -\lambda_1 - \mu_{sh}, & y=0, \\ & 0<z<K_4; \\ -\mu_{sh}, & y=0, z=K_4, \\ -\lambda_1 - \mu_{sp}, & 0<y\le K_2, \\ & z=0; \\ -\lambda_1 - \mu_{sh} - \mu_{sp}, & 0<y<K_2, \\ & 0<z<K_4; \\ -\mu_{sh} - \mu_{sp}, & 0<y<K_2, \\ & z=K_4; \\ -\lambda_1 - \mu_{sh}(1-\beta) - \mu_{sp}, & y=K_2, \\ & 0<z<K_4; \\ -\mu_{sh}(1-\beta) - \mu_{sp}, & y=K_2, z=K_4; \\ 0, & \text{otherwise.} \end{cases}$$

Elements of matrix A_2: The sub-matrix A_2 represents the phase distribution of the controller, Class CP, Class HP, and specialised hardware when the number of packets in the internal buffer (i.e. $n_b(t)$ or v in Eq. (5.12)) decreases by 1:

$$A_{2(w,w')} = \begin{cases} A_{21}{}^{(w)}, & w' = w, \\ 0, & \text{otherwise,} \end{cases}$$

where,

$$A_{21}{}^{(w)}{}_{(x,x')} = \begin{cases} A_{212}{}^{(x)}, & x' = x - 1, \\ 0, & \text{otherwise}, \end{cases}$$

where,

$$A_{212}{}^{(x)}{}_{(y,y')} = \begin{cases} A_{2121}{}^{(y)}, & y' = y, \\ 0, & \text{otherwise}. \end{cases}$$

where,

$$A_{2121}{}^{(y)}{}_{(z,z')} = \begin{cases} \mu_{sp}, & z' = z, \\ 0, & \text{otherwise}. \end{cases}$$

5.2.2 Network performance metrics for HPI

With these sub-block matrices and matrix geometric solution, we can compute the stationary distribution probabilities $(\pi_{v,w,x,y,z})$ for v packets in the internal buffer, w packets in controller, x packets in Class CP, y packets in Class HP, and z packets in the switch hardware. The performance metrics like the average packet transfer delay and packet loss probability can be computed using $\pi_{v,w,x,y,z}$.

Throughput of the internal buffer (denoted by T_{ib}) is the sum of probabilities that the internal buffer at the CPU has at least one data packet to forward with service rate of μ_{sp} for HPI. Like SPI, throughputs of the Class CP (T_{cp}) and the internal buffer (T_{ib}) for HPI are the same and this is given by:

$$T_b = T_{cp} = \mu_{sp} \sum_{v=1}^{K_3} \sum_{w=0}^{v-1} \sum_{y=0}^{K_2} \sum_{z=0}^{K_4} \pi_{v,w,x,y,z}. \tag{5.13}$$

Throughput of the controller (denoted by T_c) is the sum of probabilities that the controller has at least one control packet to forward with service rate of μ_c, and there is at least one data packet temporarily buffered in the internal buffer for HPI. This is given by:

$$T_c = \mu_c \sum_{v=1}^{K_3} \sum_{w=1}^{v} \sum_{y=0}^{K_2} \sum_{z=0}^{K_4} \pi_{v,w,x,y,z}. \tag{5.14}$$

Throughput of the Class HP (denoted by T_{hp}) is the sum of probabilities that the Class HP of the CPU buffer has at least one data packet to forward with service rate of μ_{sp} and there is no packet in the Class CP for HPI, and this is given by:

$$T_{hp} = \mu_{sp} \sum_{v=0}^{K_3} \sum_{w=0}^{v} \sum_{y=1}^{K_2} \sum_{z=0}^{K_4} \pi_{v,w,x,0,z} \quad . \tag{5.15}$$

Throughput of the specialised hardware (denoted by T_{sh}) is the sum of probabilities that the specialised hardware switch has at least one data packet to forward with service rate of μ_{sh} for HPI and this is given by:

$$T_{sh} = \mu_{sh} \sum_{v=0}^{K_3} \sum_{w=0}^{v} \sum_{y=0}^{K_2} \sum_{z=1}^{K_4} \pi_{v,w,x,y,z} \quad . \tag{5.16}$$

Average number of data packets (denoted by L_{HPI}) is the average number of data packets in HPI where data packets travel only through the specialised hardware (i.e TCAM) and the CPU (i.e the Class HP and the internal buffer) of the switch. Therefore, L_{HPI} is expressed as:

$$L_{HPI} = \sum_{v=0}^{K_3} \sum_{w=0}^{v} \sum_{y=0}^{K_2} \sum_{z=0}^{K_4} (v+y+z)\pi_{v,w,x,y,z}. \tag{5.17}$$

Average data packet transfer delay (denoted by t_{HPI}) is the mean sojourn time of a data packet in an SDN network with a single controller and hardware switch with the internal buffer. The average packet transfer delay of a data packet in HPI is obtained by applying Little's theorem to Eq. (5.17) which is expressed as:

$$t_{HPI} = L_{HPI}/T_{HPI}, \tag{5.18}$$

where T_{HPI} is the throughput of HPI expressed as:

$$T_{HPI} = T_{ib} + (1 - \beta)T_{sh}. \tag{5.19}$$

Packet loss probability (denoted by PL_{HPI}) is the total average packet loss probability for Model HPI. Assuming independence of packet arrivals between the Class CP, the Class HP, the internal buffer, and the specialised hardware queue, the packet loss probabilities of the Class CP (PL_{cp}), Class HP (PL_{hp}), the switch's internal buffer (PL_{ib}) and the specialised hardware queue (PL_{sh}) represent the average number of packets being blocked or dropped by the Class CP, Class HP, the switch's internal buffer, and the specialised hardware queue out of total incoming packets in the respective queue. The packet loss probabilities PL_{cp}, PL_{hp}, PL_{ib}, and PL_{sh} for HPI are expressed as,

$$\begin{aligned} PL_{cp} = PL_{ib} &= 1 - T_{cp}/T_c, \\ PL_{hp} &= 1 - T_{hp}/T_{sh}, \\ PL_{sh} &= 1 - T_{sh}/\lambda_1. \end{aligned} \tag{5.20}$$

Therefore, the total packet loss probability for HPI (PL_{HPI}) is the sum of packet loss probabilities in the Class CP, the Class HP, the internal buffer, and the specialised hardware queue of the switch which is given as,

$$PL_{HPI} = PL_{cp} + PL_{hp} + P_{ib} + PL_{sh}. \tag{5.21}$$

5.2.3 Buffer dimensioning for HPE and HPI

In this section, the buffer dimensioning for HPE and HPI is discussed which assumes that the switch queues are M/M/1 (see Section 3.2) as opposed to GI/M/1/K and M/M/1/K (used for specialised hardware).

The minimum queue capacity for the switch in HPE is denoted by K_{HPE} which is the sum of K_1 (i.e. minimum buffer capacity required for

Class CP), K_2 (i.e. minimum buffer capacity required for Class HP), and K_4 (i.e.minimum buffer capacity required for specialised hardware) which are calculated using the approximation in Eq. (3.2) as:

$$K_1 \geq \frac{\log[PER]}{\log[\rho_{cp}]}, K_2 \geq \frac{\log[PER]}{\log[\rho_{hp}]}, K_4 \geq \frac{\log[PER]}{\log[\rho_{sh}]}, \tag{5.22}$$

where ρ_{cp}, ρ_{hp}, and ρ_{sh} are the server utilisation at the Class CP, Class HP, and the specialised hardware, respectively, which are defined as:

$$\rho_{cp} = \frac{\beta\lambda_1}{\mu_{sp}}, \qquad \rho_{hp} = \frac{\beta\lambda_1}{\mu_{sp}}, \qquad \rho_{sh} = \frac{\lambda_1}{\mu_{sh}}.$$

Therefore, K_{HPE} can be expressed as

$$K_{HPE} = K_1 + K_2 + K_4. \tag{5.23}$$

Likewise, for HPI, the minimum buffer capacities for the Class CP, the Class HP, the internal buffer, and the specialised hardware are denoted as K_1, K_2, K_3, and K_4, respectively, and can be calculated using Eq. (3.2) as:

$$\begin{gathered} K_1 \geq \frac{\log[PER]}{\log[\rho_{cp}]}, K_2 \geq \frac{\log[PER]}{\log[\rho_{hp}]}, K_3 \geq \frac{\log[PER]}{\log[\rho_{ib}]}, \\ K_4 \geq \frac{\log[PER]}{\log[\rho_{sh}]}, \end{gathered} \tag{5.24}$$

where ρ_{cp}, ρ_{hp}, ρ_{ib}, and ρ_{sh} are server utilizations at the Class CP, the Class HP, the internal buffer of the CPU, and the specialised hardware, respectively, which are defined as:

$$\rho_{cp} = \frac{\beta\lambda_1}{\mu_{sp}}, \rho_{hp} = \frac{\beta\lambda_1}{\mu_{sp}}, \rho_{ib} = \frac{\beta\lambda_1}{\mu_{sp}}, \rho_{sh} = \frac{\lambda_1}{\mu_{sh}}.$$

Therefore, the minimum buffer capacity for the switch in HPI is the sum of minimum buffer capacity for the Class CP, the Class HP, the internal buffer, and the specialised hardware:

$$K_{HPI} = K_1 + K_2 + K_3 + K_4. \tag{5.25}$$

In this chapter, the minimum buffer capacities of the switch for HPE and HPI are K_{HPE} and K_{HPI}, respectively.

5.3 Results

This section presents the analytical and discrete event simulation results of the developed queueing models for hardware-based SDN switches with and without the internal buffer (i.e. HPE and HPI respectively).
This section is briefly categorised into the following subsections:

- ***Validating queueing models HPE & HPI*** where analytical results are compared with discrete event simulation results.
- ***Software vs. Hardware Switch*** where SDN software and hardware switches with and without the internal buffer are compared (i.e. SPE vs HPE and SPI vs HPI).
- ***HPE vs. HPI*** where SDN hardware switches with and without internal buffer are compared.

The parameters used for analysis and simulation are shown in Table 5.3. From Table 5.3, the table miss probability β varies from 0.1 to 1, the switch processor or CPU processing rate (μ_{sp}) is assumed to be 1000 packets/sec, the controller to switch processing ratio (m_r) varies from 0.1 to 2, and the specialised hardware to CPU processing ratio (m_s) varies from 100 to 1000. The external arrival rate (λ_1) to switch from each host is assumed to be 120 or 240 or 480 packets/sec and we assume an Ethernet network for which the BER is assumed to be 10^{-12}. The number of hosts per switch (N) varies from 1 to 80 to study the effect of the internal buffer in SDN hardware switches.

The discrete event simulations based on Monte Carlo simulation for models HPE and HPI with pseudo codes are explained in Appendix F and Appendix G, respectively. The simulations are repeated 100 times and the 95% confidence intervals (CI) are computed on the basis that the errors are normally distributed.

In the following subsections, to take into consideration packet loss probability, the queue capacities of the Class ES (in SPE and SPI), the Class HP

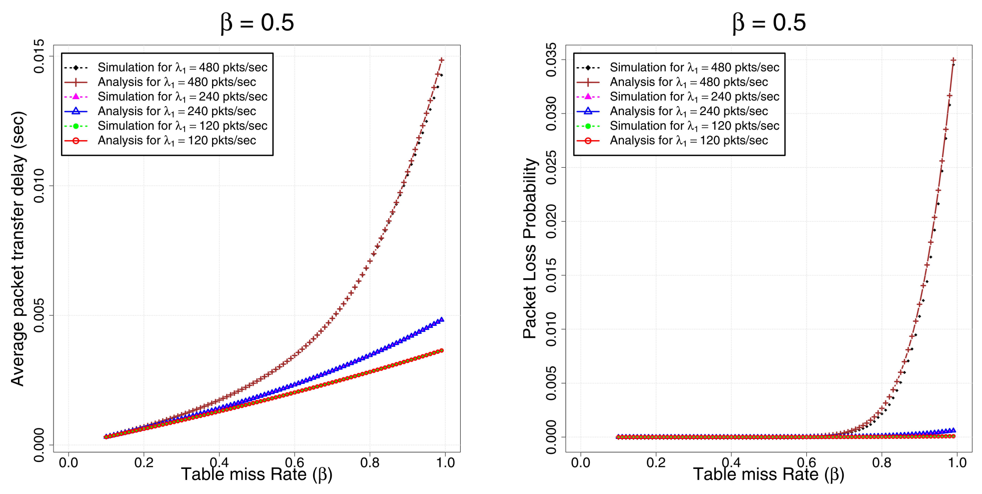
Table 5.3: Parameter used for analysis and simulation for both HPE and HPI.

Parameter	Value
Table miss probability, β	0.1$\sim$1
CPU processing rate, μ_{sp} (packets/sec)	1000
Controller to CPU Processing Ratio (μ_c/μ_{sp}), m_r	0.1$\sim$2
Specialised hardware to CPU Processing Ratio (μ_{sh}/μ_{sp}), m_s	100$\sim$1000
Arrival rate, λ_1 (packets/sec)	120, 240, 480
Bit Error Rate, BER	10^{-12}
MTU TCP packet size (byte)	1500
Number of hosts per switch, N	$1 \sim 80$

(in HPE and HPI), and the specialised hardware queue (in HPE and HPI) are assumed to be half of their minimum queue capacities determined from buffer dimensioning (using Eq. 5.23 and Eq. 5.25). The queue capacities of the Class CS (in SPI), the Class CP (in HPI), and the internal buffer (in both SPI and HPI) are minimum queue capacities determined from buffer dimensioning where there is no packet loss. This buffer sizing ensures no loss of control packets.

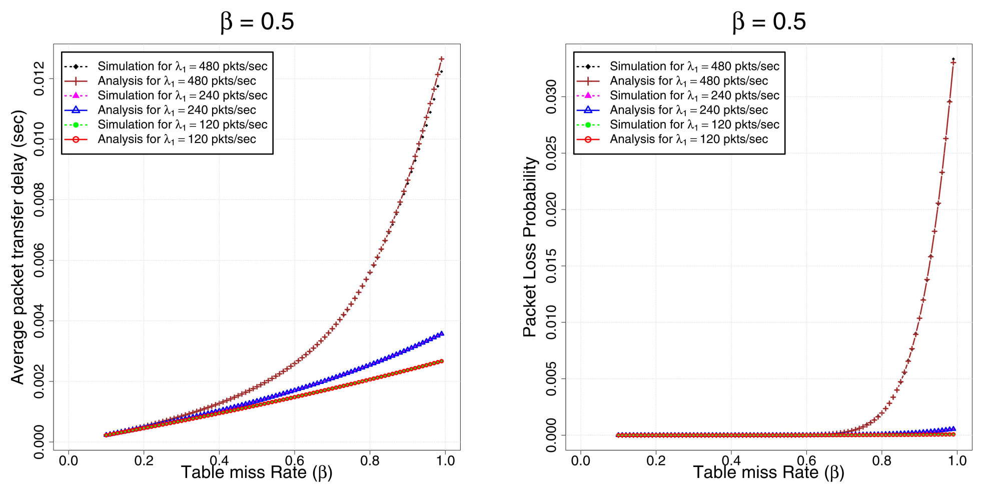
5.3.1 Validating queueing models HPE & HPI

The validation of analytical results for HPE and HPI is done by comparing them with discrete event simulation results. Figures 5.4 and 5.5 show the validation results for HPE and HPI respectively for increasing β with $m_r = 1$ and $m_s = 1000$. The error percentage between analysis and simulation predictions for both average packet transfer delay and packet loss probability is between 0.6%-2.8% as shown in Fig. 5.4 and Fig. 5.5. The error for HPI is slightly higher than HPE by 1%. This range of error is acceptable for analysis, as computation of π distributions for HPI which is modelled as a nonhomogenous QBD process is prone to inaccuracy due to the possibility of singular matrix becoming nonsingular in machine preci-



(a) (b)

Figure 5.4: Validation of Model HPE in terms of (a) Average packet transfer delay, and (b) Packet loss probability.



(a) (b)

Figure 5.5: Validation of Model HPI in terms of (a) Average packet transfer delay, and (b) Packet loss probability.

sion [85].

5.3.2 Software vs. Hardware switch

5.3.2.1 Relative minimum buffer capacity

In this section, the the relative minimum buffer capacity between a software and hardware switch are compared as seen in Fig. 5.6.

The relative minimum buffer capacity between a software and hardware switch without the internal buffer (SPE and HPE) is denoted as ϵ_{Ka} and defined as,

$$\epsilon_{Ka} = \frac{K_{\text{HPE}} - K_{\text{SPE}}}{K_{\text{SPE}}} \times 100\%.$$

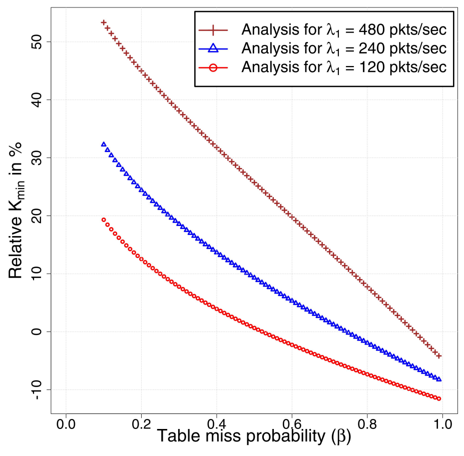
Similarly, the relative minimum capacity between a software switch and hardware switch with the internal buffer (SPI and HPI) is denoted as ϵ_{Kb} and defined as,

$$\epsilon_{Kb} = \frac{K_{\text{HPI}} - K_{\text{SPI}}}{K_{\text{SPI}}} \times 100\%.$$

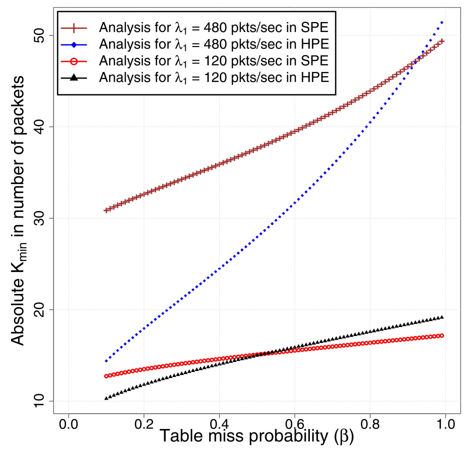
Positive values of ϵ_{Ka} and ϵ_{Kb} mean SPE and SPI require less capacity than HPE and HPI respectively, while a negative value implies SPE and SPI require more capacity than HPE and HPI respectively.

Figure 5.6a shows the ϵ_{Ka} curves and Fig. 5.6b shows the absolute values of K_{SPE} and K_{HPE} for increasing β. Similarly, Fig. 5.6c shows the ϵ_{Kb} curves and Fig. 5.6d shows the absolute values of K_{SPI} and K_{HPI} for increasing β. From Fig. 5.6a, it can be observed that HPE requires up to 55% more buffer capacity than SPE. This number is slighty lower for HPI which requires up to 45% more buffer capacity than SPE as seen in Fig. 5.6c.

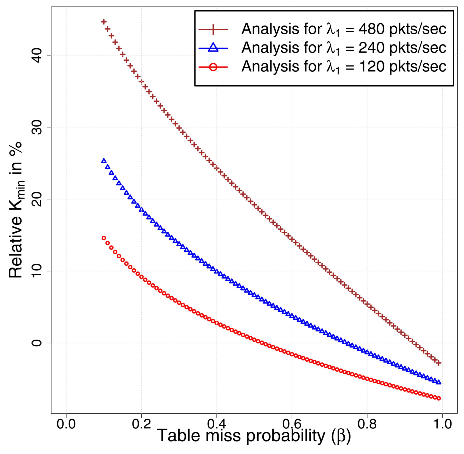
This is because the switches in both SPI and HPI require queue capacities for the CPU, the specialised hardware, and the internal buffer, while the switches in SPE and HPE require queue capacities for the CPU and specialised hardware only. Given the fact that relative differences $(K_{\text{SPE}} - K_{\text{HPE}})$ and $(K_{\text{SPI}} - K_{\text{HPI}})$ are the same irrespective of with or without the internal buffer but relative parameters K_{SPE} and K_{SPI} are different.



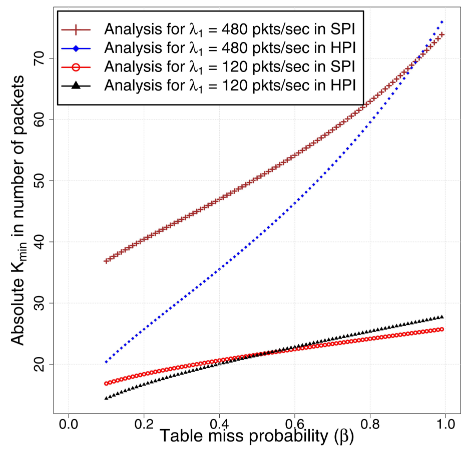
(a)



(b)



(c)



(d)

Figure 5.6: Minimum switch queue capacity (K_{min}) (a) Relative difference between SPE and HPE i.e. ϵ_{Ka}, (b) Absolute value between SPE and HPE, (c) Relative difference between SPI and HPI i.e. ϵ_{Kb} and (d) Absolute value between SPI and HPI for $\mu_{sp} = 1000$ pkts/sec and increasing β.

It is evident that a higher value of the relative parameter gives a lower relative minimum queue capacity, which in this case K_{SPI} has a higher value than K_{SPE} due to the internal buffer.

5.3.2.2 Relative average delay

In this section, the relative average packet transfer delay between software and hardware switch are compared as seen in Fig. 5.7.

SPE vs. HPE: In this comparison, effects of the delay in a software and hardware switch without the internal buffer is compared. In Fig. 5.7a and Fig. 5.7b, the average delay in the switch between SPE (denoted by t_{SPE} as in Eq. (3.30)) and HPE (denoted by t_{HPE} as in Eq. (5.7)) is compared for increasing β and m_r, respectively.

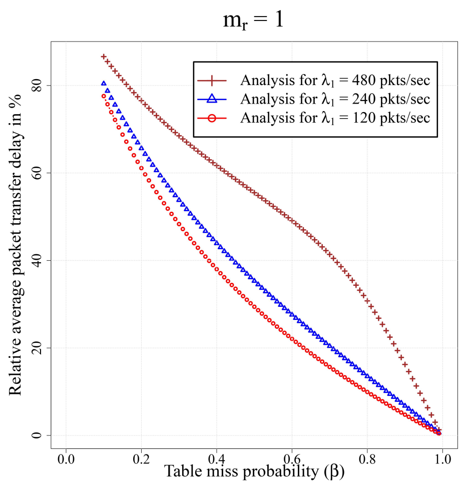
The relative time difference (denoted by ϵ_{da}) to traverse packets in the switch between SPE and HPE (both with finite capacity) is calculated as:

$$\epsilon_{da} = \frac{(t_{SPE} - t_{HPE})}{t_{SPE}} \times 100\%.$$

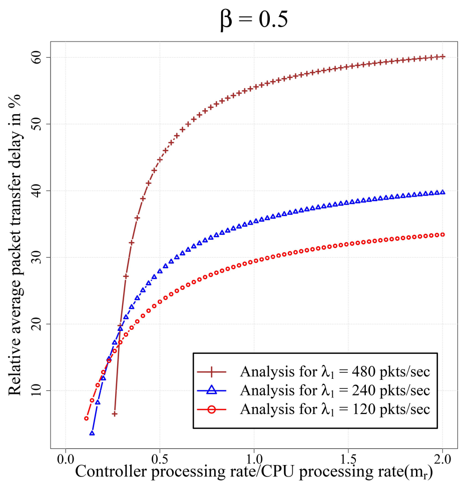
A positive value of ϵ_{da} means HPE has lower average delay for a packet to travel in the network compared to SPE.
Figure 5.7a and Fig. 5.7b show the average relative time for packets to traverse a switch between SPE and HPE for varying β and m_r respectively. From Fig. 5.7a, the relative average packet delay decreases from 85% to 1.8% for increasing β. However, the trend is reversed in Fig. 5.7b with increasing m_r, whereby the relative total delay increases from 0.8% to 56% as m_r increases from 0.5 to 2.0.

As the value of β increases, the traffic processed by CPU in Model HPE significantly increases, therefore increasing the delay predicted by HPE and diminishing the benefit of having dedicated switch hardware for forwarding.

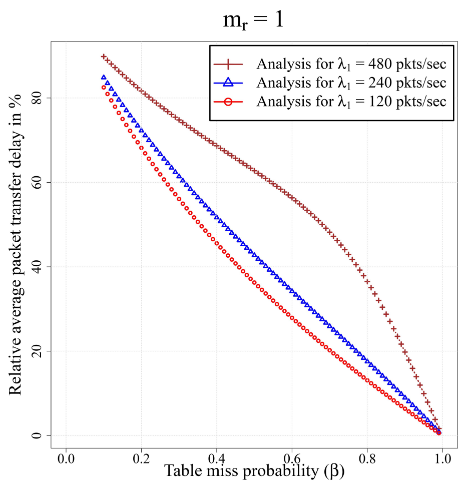
Moreover, as m_r increases, the relative average delay between HPE and SPE increases exponentially and then plateaus off. The reason for this is



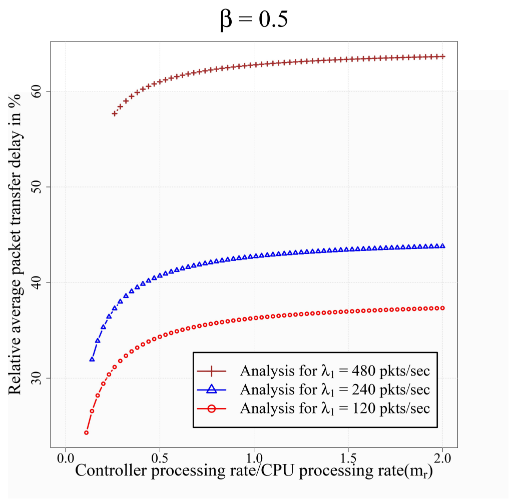
(a)



(b)



(c)



(d)

Figure 5.7: Relative average packet transfer delay between (a – b) SPE and HPE in % i.e. ϵ_{da} , (c – d) SPI and HPI in % i.e. ϵ_{db} for increasing β and m_r.

that higher m_r means controller processing power is higher than CPU processor at the switch which significantly reduces the delay caused by packets traversing the control path.

SPI vs. HPI: This comparison helps to investigate the effect of the internal buffer in a software and hardware switch with reference to average packet transfer delay. The average delay in the switch between SPI (denoted by t_{SPI} as in Eq. (4.6)) and HPI (denoted by t_{HPI} as in Eq. (5.18)) for increasing β and m_r is shown in Fig. 5.7c and Fig. 5.7d, respectively.

The relative average packet transfer delay (denoted by ϵ_{db}) between SPI and HPI (both with finite capacity) is calculated as:

$$\epsilon_{db} = \frac{(t_{SPI} - t_{HPI})}{t_{SPI}} \times 100\%.$$

A positive value of ϵ_{db} means HPI has a lower average delay for a packet to travel in the network compared to SPI.
Figure 5.7c and Fig. 5.7d show the average relative time for packets to traverse a switch between SPI and HPI for varying β and m_r respectively. From Fig. 5.7c, the relative average packet delay in a switch with the internal buffer is same as that of a switch without the internal buffer for increasing β. This is because the delay caused by a faster controller has insignificant effects on switches with and without the internal buffer.

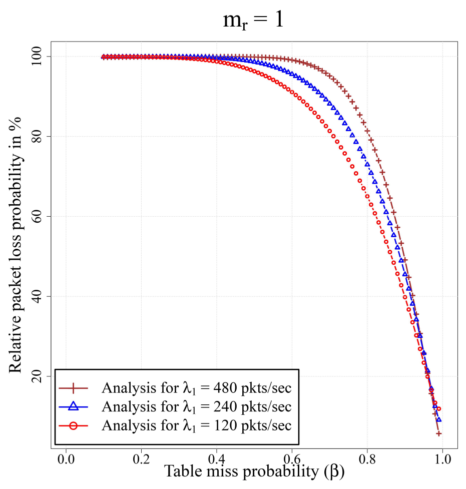
However, the relative delay between SPI and HPI is different from SPE and HPE for the slower controller as seen in Fig. 5.7d, whereby the relative total delay increases from 25%. This is because the internal buffer causes a reduction in the size of packet-in messages that are sent to the controller.

This shows the benefit of a hardware switch over a software switch, that significantly reduces the overall average delay of the packet for a lower β and higher m_r.

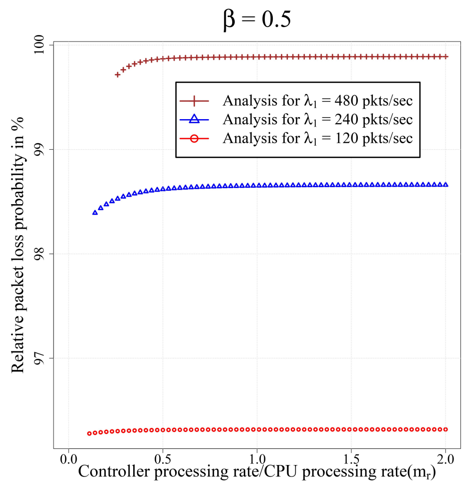
5.3.2.3 Relative average packet loss probability

In this section, the packet loss probability between a software and hardware switch are compared as seen in Fig. 5.8.

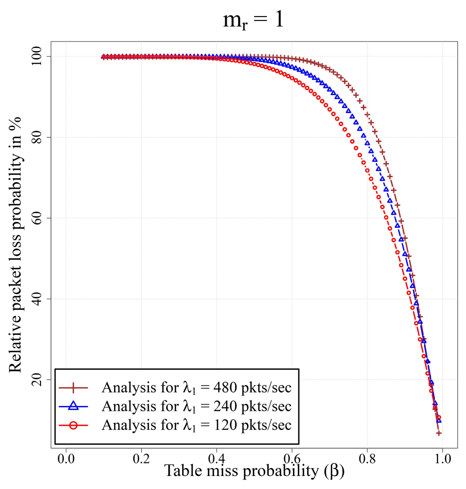
SPE vs. HPE: In this comparison, effects of the packet loss probability in a software and hardware switch without the internal buffer are investi-



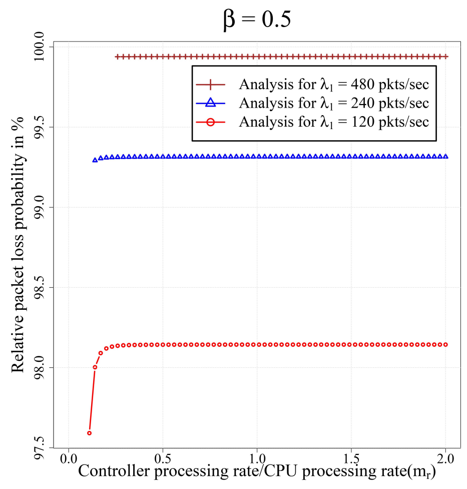
(a)



(b)



(c)



(d)

Figure 5.8: Relative average packet loss probability between (a – b) SPE and HPE i.e. ϵ_{la}, (c – d) SPI and HPI i.e. ϵ_{lb} for increasing β and m_r.

gated. In Fig. 5.8a and Fig. 5.8b, the packet loss probability in the switch between SPE (denoted by PL_{SPE} as in Eq. (3.33)) and HPE (denoted by PL_{HPE} as in Eq. (5.10)) is compared for increasing β and m_r, respectively.

The relative packet loss probability difference (denoted by ϵ_{la}) in the switch between SPE and HPE (both with finite capacity) is calculated as:

$$\epsilon_{la} = \frac{(PL_{SPE} - PL_{HPE})}{PL_{SPE}} \times 100\%.$$

A positive value of ϵ_{la} means HPE has a lower packet loss rate compared to SPE.

From Fig. 5.8a, the relative packet loss probability is up to 100% higher in Model SPE than Model HPE for lower β, and it decreases to approximately 2% for $\beta = 1.0$. The reason for this is, as the β increases, the amount of traffic processed by the CPU in Model HPE significantly increases (i.e. the hardware processing at the switch is not leveraged) rendering its performance closer to that of SPE, therefore reducing the gap between models SPE and HPE for packet loss probability. This difference in performance decreases drastically for higher β, for which the packet loss probability is much higher due to increasing amount of traffic to be processed by the CPU.

Similarly, the relative packet loss probability is up to 83% higher in Model SPE than Model HPE for lower m_r and remains a steady 100% with increasing m_r as seen in Fig. 5.8b. The reason for the increasing relative packet loss probability with increasing m_r is: a higher m_r means higher controller processing power than CPU processor which quickly feeds back the packet to the CPU, significantly increasing the packet loss probability in the CPU.

SPI vs. HPI: In this comparison the effect of the internal buffer in a software and hardware switch with reference to the packet loss probability is investigated. In Fig. 5.8c and Fig. 5.8d, the packet loss probability in the switch between SPI (denoted by PL_{SPI} as in Eq. (4.8)) and HPI (denoted by PL_{HPI} as in Eq. (5.21)) is compared for increasing β and m_r, respectively.

The relative packet loss probability difference (denoted by ϵ_{lb}) in the switch between SPI and HPI (both with finite capacity) is calculated as:

$$\epsilon_{lb} = \frac{(PL_{SPI} - PL_{HPI})}{PL_{SPI}} \times 100\%.$$

A positive value of ϵ_{lb} means HPI has lower packet loss rate compared to SPI.

The trend for the relative packet loss probability in Fig. 5.8c is similar to the packet loss probability in Fig 5.8a for varying β. The reason for this is, a faster controller introduces lower delay that has insignificant effects on switches with and without the internal buffer.

Similarly, the relative packet loss probability between SPI and HPI (as seen in Fig. 5.8d) follows a similar trend to that of the relative packet loss probability between SPE and HPE (as seen in Fig 5.8b), except for lower m_r. This is because with a slower controller, the waiting time of packets at the switch increases, increasing the contention rate and blocking of packets. The contention for the buffer space caused by a slower controller is reduced with internal buffering, hence reducing the blocking of packets compared to the switch without internal buffer.

This shows the benefit of a hardware switch over a software switch, that significantly reduces the packet loss probability.

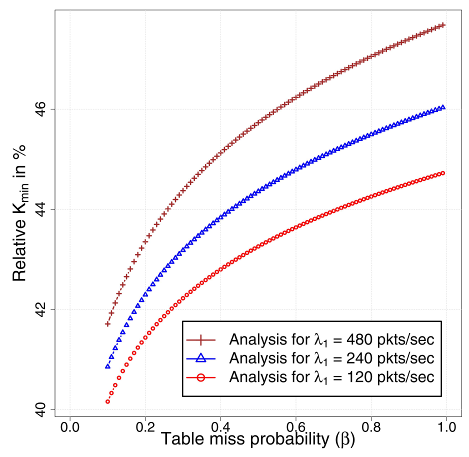
5.3.3 HPE vs. HPI

5.3.3.1 Relative minimum buffer capacity

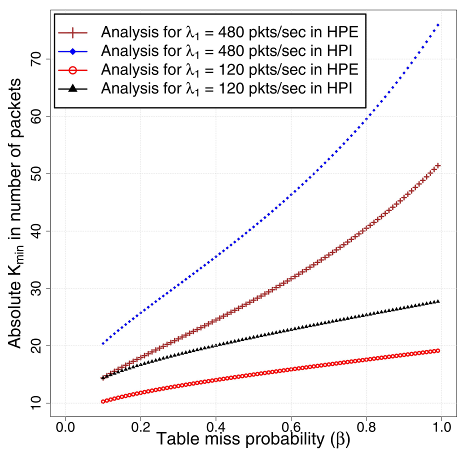
In this section, the relative minimum buffer capacity between a hardware switch with (i.e. HPI) and without the internal buffer (i.e. HPE) is computed. The relative minimum capacity between HPE and HPI is denoted as ϵ_{Kc} and defined as,

$$\epsilon_{Kc} = \frac{K_{\text{HPI}} - K_{\text{HPE}}}{K_{\text{HPI}}} \times 100\%.$$

A positive value of ϵ_{Kc} means HPE requires less capacity than HPI, while a negative value implies HPE requiring more capacity than HPI.



(a)



(b)

Figure 5.9: Minimum switch queue capacity (K_{min}) between HPE and HPI for $\mu_{sp} = 1000$ pkts/sec and increasing β: (a) Relative difference i.e. ϵ_{Kc} and (b) Absolute value.

Figure 5.9 shows the ϵ_{Kc} curve for increasing β. From Fig. 5.9, it can be observed that HPI requires upto 50% more buffer capacity than HPE.

This is because the switch in HPI requires buffer capacities for the CPU, the specialised hardware, and the internal buffer. While, the switch in HPE requires buffer capacities for the CPU and specialised hardware only.

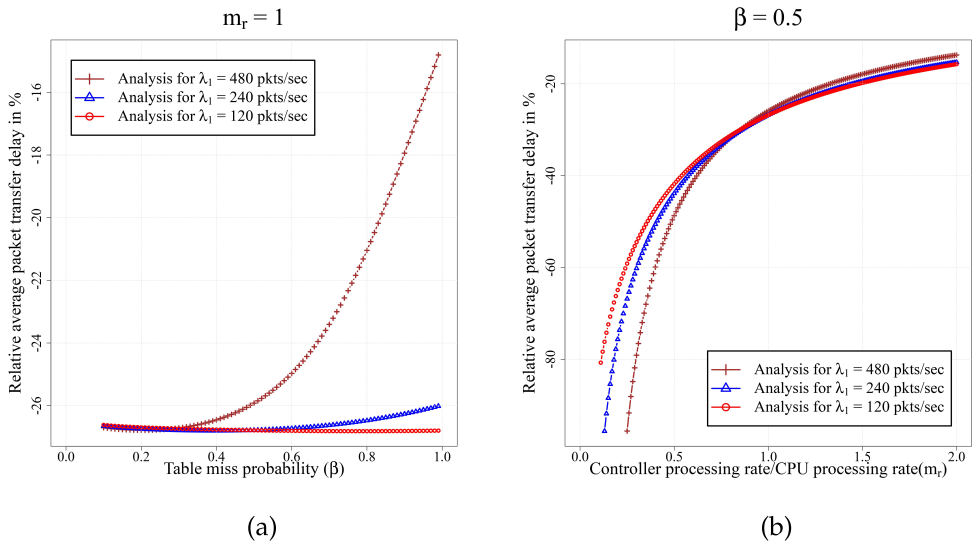
5.3.3.2 Relative average delay

In this section, the average packet transfer delay between HPE (denoted by t_{HPE} as in Eq. (5.7)) and HPI (denoted by t_{HPI} as in Eq. (5.18)) is compared. This comparison helps to investigate the effect of the internal buffer in a hardware switch with reference to the average packet transfer delay.

The relative average packet transfer delay (denoted by ϵ_{dc}) between HPE and HPI (both with finite capacity) is calculated as:

$$\epsilon_{dc} = \frac{(t_{HPI} - t_{HPE})}{t_{HPE}} \times 100\%.$$

A positive value of ϵ_{dc} means HPE has a lower average delay for a packet to travel in the network compared to HPI.



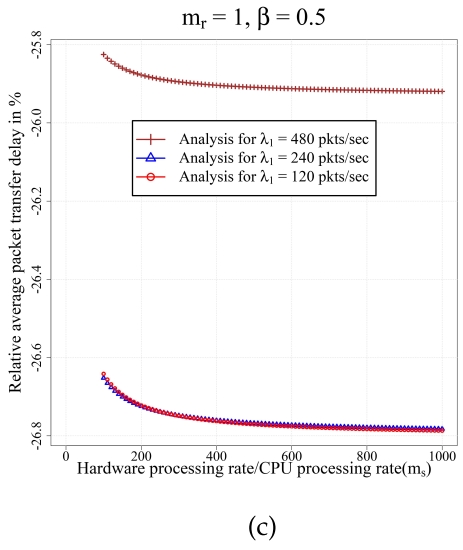


Figure 5.10: Relative average packet transfer delay between HPE and HPI i.e. ϵ_{dc} for (a) increasing β and $m_r = 1$, $m_s = 1000$; (b) increasing m_r and $\beta = 0.5$, $m_s = 1000$; and (c) increasing m_s and $m_r = 1$, $\beta = 0.5$.

Figure 5.10 shows the relative average packet transfer delay between

HPE and HPI in percentile. Figures 5.10a, 5.10b, and 5.10c show the relative average delay for increasing β, m_r, and m_s respectively. From Figs. 5.10a, 5.10b, and 5.10c, it can be observed that HPI exhibits up to 28%, 80% and 26% reduction in the average delay of a packet compared to HPE respectively.

This is because with increasing β the number of packets sent to the controller is increased. These packets requires a decision from the controller are processed by the CPU which has a slower processor than the specialised hardware. With internal buffering, the CPU absorbs these packets with temporary buffering and encapsulates packet-in message with a small part of the data packets (mostly packet header information). The smaller packet-in messages are processed faster by the controller than packet-in messages encapsulated with a full data packet as in the case of a hardware switch without the internal buffer. The temporary buffering significantly reduces the overall delay for increasing traffic forwarded by the specialised hardware as seen in Fig. 5.10a.

Similarly, with the increasing m_r the controller processing capacity increases. The performance of the controller degrades with increasing flow update requests from the switch. Such a controller can be called a slower controller and can be related with a lower value of m_r. The real benefit of the internal buffer at the switch can be observed for a slower controller as seen in Fig. 5.10b . With internal buffering of packets awaiting decision from the controller, the control traffic size can be reduced, allowing the controller to process packets faster.

Also, with the increasing hardware processing capacity, the average delay of packets at the specialised hardware is reduced as seen in Fig. 5.10c. Again with internal buffering, the overall average delay is reduced as compared to without internal buffering.

This shows the benefit of a hardware-based SDN switch with the internal buffer over a hardware-based SDN switch without the internal buffer, that significantly reduces the overall average delay of the packet for lower

β and m_r.

5.3.3.3 Relative average packet loss probability

In this section, the average packet loss probability between HPE (denoted by PL_{HPE} as in Eq. (5.10)) and HPI (denoted by PL_{HPI} as in Eq. (5.21)) is compared. This comparison helps to investigate the effect of the internal buffer in a hardware switch with reference to the packet loss probability.

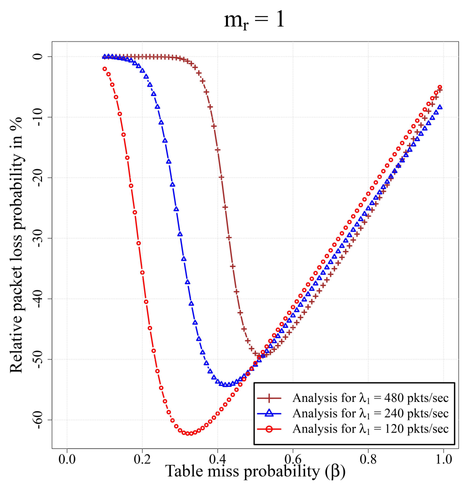
The relative packet loss probability (denoted by ϵ_{lc}) between HPE and HPI (both with finite capacity) is calculated as:

$$\epsilon_{lc} = \frac{(PL_{HPI} - PL_{HPE})}{PL_{HPE}} \times 100\%.$$

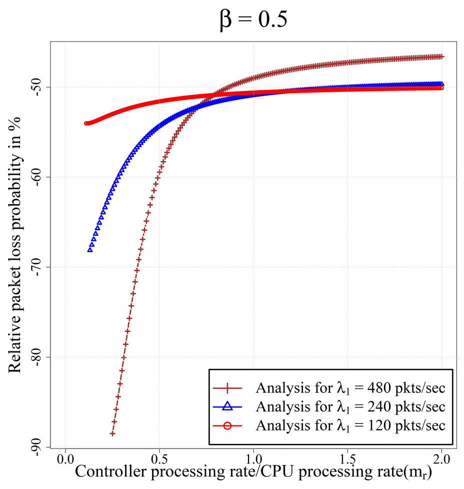
A positive value of ϵ_{lc} means HPE has a lower packet loss probability compared to HPI.

Figure 5.11 shows the relative packet loss probability between HPE and HPI in percentile. Figure 5.11a, Fig. 5.11b, and Fig. 5.11c show the relative packet loss probability for increasing β, m_r, and m_s respectively. In Fig. 5.11a, HPI exhibits up to 60% reduction in the packet loss probability for a lower β and up to 6% reduction for a higher β. This is due to a limited memory resource available for an internal buffering. With a lower β, the number of data packets to be stored in the internal buffer is lower which gives minimal packet loss in HPI. Whereas with a higher β, the number of data packets to be stored in the internal buffer can be higher than the buffer capacity of the internal buffer which gives significant packet loss in HPI.

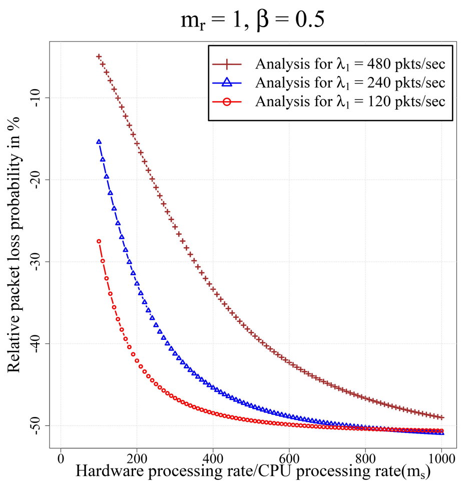
Similarly, in Fig. 5.11b, HPI exhibits up to 89% reduction in the packet loss probability for lower m_r and up to 50% reduction for higher m_r. It is observed that higher value of λ_1 (i.e 480 pkts/sec) significantly reduces the packet loss probability for a lower value of m_r (i.e $m_r \lessapprox 1$), whereas the lower value of λ_1 (i.e 120 or 240 pkts/sec) shows a better packet loss probability for a higher value of m_r (i.e $m_r \gtrapprox 1$). This is because lower m_r means the controller is slower than the CPU and higher m_r means the



(a)



(b)



(c)

Figure 5.11: Relative average packet loss probability between HPE and HPI i.e. ϵ_t for (a) increasing β and $m_r = 1$, $m_s = 1000$; (b) increasing m_r and $\beta = 0.5$, $m_s = 1000$; and (c) increasing m_s and $m_r = 1$, $\beta = 0.5$.

controller is faster than the CPU. The slower controller processes packet-in messages slower, which increases the blocking of data packets in the CPU. With the internal buffer, the blocking of data packets in the CPU is significantly reduced, whereas the faster controller processes the packet-in messages faster, which reduces the blocking of data packets in the CPU. This reduces the benefit of the internal buffer especially for a higher value of λ_1.

Similarly, with the increasing hardware processing time, HPI exhibits a lower packet loss probability than HPE as seen in Fig. 5.11c. This is because the average waiting time of packets at the specialised hardware reduces with the increasing hardware processing time. This lower waiting time of packets reduces the packet loss at the switch hardware. Furthermore, internal buffering reduces the packet loss by temporarily buffering packets at the CPU.

This shows the benefit of a hardware switch with the internal buffer over a hardware switch without the internal buffer for the lower β and m_r, which significantly reduces the packet loss probability.

5.3.3.4 Effect of varying number of hosts connected to switch

So far, the external traffic at the specialised hardware of the switch is assumed to arrive from a single host (i.e. $N = 1$). In this section, the effect of varying the number of hosts for both HPE and HPI is presented by varying N from 1 to 80 with the following assumptions:

(i) The minimum queue capacities for the CPU, the internal buffer, and the specialised hardware in both HPE and HPI remain the same for a varying number of hosts.

(ii) Similarly, the value of processing time for the CPU, specialised hardware in the switch, and controller remains the same for a varying number of hosts.

(iii) The external traffic (λ_1) arriving at the specialised hardware from each host is assumed to be homogeneous.

(iv) The effective external arrival rate (λ_{eff}) at the switch from N hosts is calculated as:

$$\lambda_{eff} = N \times \lambda_1. \tag{5.26}$$

(v) Finally, taking into account of a computation complexity due to limited memory resources in a computation device, the initial value of λ_1 is taken as 12 pkts/sec. This value of λ_1 is also the reason for the maximum number of hosts being selected as 80. The effective arrival rate for $N(=80)$ hosts gives 960 pkts/sec (80×12 pkts/sec) which is less than μ_{sp} of 1000 pkts/sec ensuring the stationary distribution condition.

Figure 5.12 shows the effect of varying the number of hosts for $\lambda_1 = 12$ pkts/sec, $m_r = 1$ and $\beta = 0.5$. Figures 5.12a and 5.12b show the effect of varying the number of hosts on the average packet transfer delay and packet loss probability, respectively. From Fig. 5.12a, with the increase in number of hosts, HPI exhibits much lower average packet transfer delay than HPE. Similarly, from Fig. 5.12b, the packet loss probability for both HPE and HPI is identical and increases with the increase in the number of switches.

This is because with the increase in number of hosts, the net arrival of packets at both HPE and HPI increases exponentially, increasing the contention rate over limited buffer capacity. The internal buffer in HPI absorbs packets requiring flow updates from the controller. This absorption of packet reduces the overall delay in HPI as compared to HPE.

5.4 Conclusion

In this chapter, queueing models for a hardware-based SDN switch with and without the internal buffer are developed, i.e. HPI and HPE, respectively. Comparative analyses were done between a software and hardware switch (a) without the internal buffer (SPE vs HPE), (b) and with the internal buffer (SPI vs HPI) in an SDN to provide insights on the performance

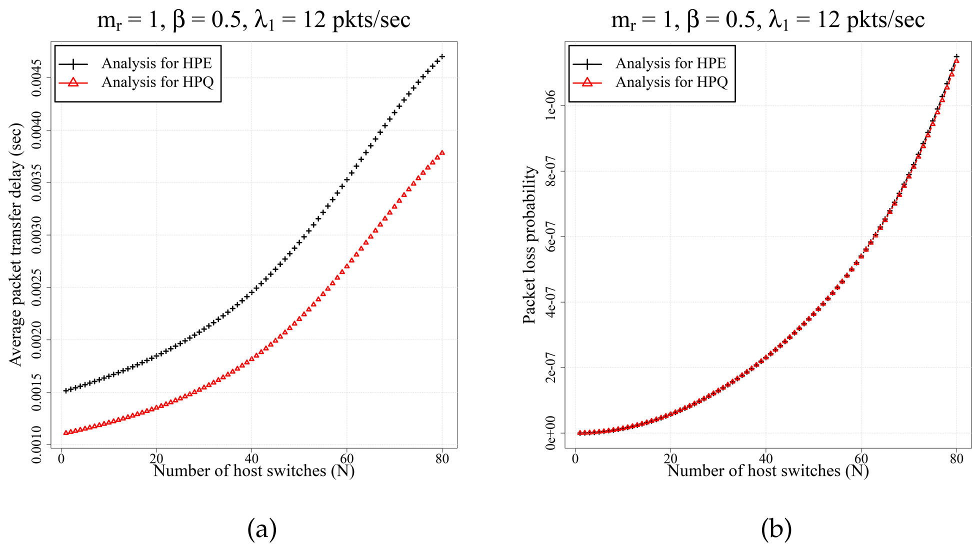


Figure 5.12: Effect of varying number of hosts for $\lambda_1 = 12$ pkts/sec, $m_r = 1$ and $\beta = 0.5$.

of software and hardware switches. Similarly, the impact of the internal buffer in a hardware-based SDN switch is also investigated (HPE vs HPI). From comparisons, the following conclusions were made:

- Software vs. Hardware switch
  - The hardware switch requires additional buffer (almost 50%) compared to the software switch,
  - The hardware switch significantly reduces the average packet transfer delay (almost 80%) compared to software switch.
  - The hardware switch significantly reduces the packet loss probability (almost 99%) compared to software switch.
- Hardware switch without vs. with the internal buffer
  - The hardware switch with the internal buffer requires almost 50% additional buffer compared to the hardware switch without the internal buffer,

- Internal buffering significantly reduces the average packet transfer delay (almost 85%) for a slower controller and (almost 20%) for a faster controller compared to without the internal buffer in a hardware switch.
- Internal buffering significantly reduces the packet loss probability (almost 60%) for a lower table miss probability and (almost 6%) for a higher table miss probability compared to without the internal buffer in a hardware switch.
- For increasing number of hosts connected to the switch, a hardware switch with the internal buffer exhibits significantly lower delay compared to a hardware switch without the internal buffer.

5.5 Summary

Based on the investigation from this chapter, the developed queueing models for a hardware-based SDN switch with and without the internal buffer perform better than software-based SDN switches in terms of the average delay and packet loss probability. The use of the internal buffer on a hardware switch reduces average delay and packet loss probability at the cost of additional memory required for internal buffering.

However there is a contingent on a stable network whereby the number of new flows that arrive at a switch for decisioning is low. Increasing involvement of the controller means that the hardware switch increasingly uses the CPU for forwarding (as opposed to ASICs and TCAM) hence reducing the benefits of using a hardware switch.

In the following chapter, this thesis is concluded with a summary and contribution of each chapters followed by future works that could not be addressed due to lack of time.

Chapter 6

Conclusions

Software-Defined Networking (SDN) simplifies the forwarding function of a switch by moving the control function away from the data plane. The performance of a switch with the data plane only is critical as it influences the overall performance of an SDN. Therefore, this thesis has focused on improving the network performance by modelling and analysing the performance of a switch in an SDN. The overall goal was to identify potential bottlenecks in a switch that can hinder the overall performance of an SDN. The three aspects of an SDN switch that needed further research to improve the performance of an SDN were identified. The three aspects are buffer sharing mechanisms, internal buffering, and SDN switch types (software and hardware). These aspects were studied via modelling and performance analysis using queueing theory.

This thesis started with the comparison of existing buffer sharing mechanisms (the shared buffer and priority buffer) for an SDN switch with the help of buffer dimensioning in Chapter 3. The buffer dimensioning is performed for the output buffer of an SDN switch to ensure packet losses are no more than a desired link loss rate.

The comparison shows that the priority buffer optimises the network performance with shorter delays when updating flow tables in an SDN switch. However, this benefit is achieved at the cost of extra buffer capacity required to ensure no packet loss in an SDN switch. Based on the findings in Chapter 3, the priority buffer is used for the output buffer of an SDN switch in the Chapters 4 and 5.

In Chapter 4, the queueing model for an SDN switch with the internal buffer is developed to investigate the impact of internal buffering. The investigation shows that the use of internal buffer reduces the overall delay and packet loss at the time of congestion. However, this benefit is achieved at the cost of additional memory required to support internal buffering.

Based on the findings from Chapter 4, queueing models for hardware-based SDN switches with and without the internal buffer are developed in Chapter 5. In Chapters 3 and 4, only software-based SDN switches are analysed. Therefore, the unified queueing model is developed in Chapter 5 to characterise the performance of SDN-based hardware and software switches. The analysis in Chapter 5 justifies the benefit of a hardware-based SDN switch with lower delay and lower packet loss through the increasing involvement of the specialised hardware for forwarding packets at the line speed rate. However, the increasing involvement of the central processing unit (CPU) for forwarding packets reduces the benefit of using a hardware-based SDN switch.

The queueing models presented in this thesis will guide network engineers and analysts to predict performance measures such as delay and loss in an SDN switch to improve the network performance.

6.1 Contributions

This thesis contributed to the performance analysis of an SDN switch by providing guidelines to switch designers and network analysers. The contributions of this thesis are listed as follows.

1. The shared buffer and the priority buffer model for the output buffer of an SDN switch were compared with buffer dimensioning. The

priority buffer model for an SDN switch accurately represents SDN behaviour by isolating control traffic and data traffic. The priority buffer model provides protection of a flow updates with a lower average time to install FTE and packet loss. Therefore, the priority buffer model will be the preferred choice for a switch in an SDN.

2. The queueing model for an SDN switch with the internal buffer is developed. This queueing model helps to investigate the potential benefits and trade-off of internal buffering in an SDN switch.
3. The queueing models for hardware-based SDN switches with and without the internal buffer at the CPU are developed. These models are used as unified queueing models for characterising the performance of a software and hardware switch in an SDN.

6.2 Future Work

The research presented in this thesis can be extended with the following future works.

Experimental Validation The queueing models for a software and hardware-based SDN switch presented in this thesis can be validated experimentally that can provide realistic detail insights.

The reason why the experimental validation could not be done in this thesis is due to the difficulty in realising priority queueing buffer in real hardware or software switches. The priority queueing buffer in this thesis uses two logical blocks to isolate control traffic from data traffic with priority. The realisation of these blocks in real hardware or software switches is not feasible at present. However, from an abstract level, these models closely match with discrete-event simulation where logical separation can be realised. This is the limitation of analytical models where the complexity of the analysis increases manifold as fewer assumptions are made, yet the analysis still remains at an abstract level when compared to real experiments.

With an experimental analysis, the difference between analytical results and experimental results may increase (say up to 10%). This is because smaller details are difficult to take into account in analytical modelling. The modelling becomes complex with too many details and therefore require assumptions. The observed difference between analytical results and discrete event simulation results are in the range of 0.2% to 2.9% [71, 126].

Partial Encapsulation of data packets In this thesis only two encapsulation methods for packet processing at an SDN switch are discussed. These two encapsulation methods are: (a) Full encapsulation of a data packet, and (b) Queueing of a data packet at the switch with sufficient memory for internal buffering as discussed in Section 2.2.1. In Section 2.2.1, the third encapsulation method i.e. partial encapsulation of a packet at the switch with limited memory to support internal buffering is also discussed. With partial encapsulation, an SDN switch will generate two types of packet-in messages to the controller :(i) small-sized packet-in message encapsulated with part of a data packet, and (ii) large-sized packet-in message encapsulated with a full data packet requiring decisions from the SDN controller.

To consider these two types of packet-in messages for the partial encapsulation method, traffic classification will be required at the controller. In addition, the switch will also require additional traffic classification to distinguish an external data traffic, a small-sized and large-sized packet-out messages feedback by the controller.

The queueing model for the partial encapsulation of a packet at the switch will give insights for switching performance of a memory-constrained CPU with support for internal buffering. Clearly, the partial encapsulation method affects the performance of the switch and controller which need to be studied and investigated.

Bibliography

[1] J. Mao, B. Han, Z. Sun, X. Lu, and Z. Zhang, "Efficient mismatched packet buffer management with packet order-preserving for OpenFlow networks," *Computer Networks*, vol. 110, pp. 91–103, 2016.

[2] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[3] B. Nunes, M. Mendonca, O. Xuan-Nam Nguyen, and T. K., "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *Communications Surveys &amp; Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[4] P. Goransson and C. Black, *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[5] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 3–14, ACM, 2013.

[6] R. Trivisonno, R. Guerzoni, I. Vaishnavi, and D. Soldani, "SDN-based 5G mobile networks: architecture, functions, procedures and backward compatibility," *Transactions on Emerging Telecommunications Technologies*, vol. 26, no. 1, pp. 82–92, 2015.

[7] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos, “Software-defined industrial internet of things in the context of industry 4.0,” *IEEE Sensors Journal*, vol. 16, no. 20, pp. 7373–7380, 2016.

[8] R. Ravindran and G.-Q. Wang, “Software-defined information centric network (ICN),” 2017. US Patent 9,838,333.

[9] K. Xu, X. Wang, W. Wei, H. Song, and B. Mao, “Toward software defined smart home,” *IEEE Communications Magazine*, vol. 54, no. 5, pp. 116–122, 2016.

[10] M.-K. Shin, K.-H. Nam, and H.-J. Kim, “Software-defined networking (SDN): A reference architecture and open APIs,” in *ICT Convergence (ICTC), 2012 International Conference on*, pp. 360–361, IEEE, 2012.

[11] Y. Goto, H. Masuyama, B. Ng, W. K. Seah, and Y. Takahashi, “Queueing analysis of software defined network with realistic openflow–based switch model,” in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pp. 301–306, IEEE, 2016.

[12] W. Miao, G. Min, Y. Wu, and H. Wang, “Performance modelling of preemption-based packet scheduling for data plane in software defined networks,” in *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on*, pp. 60–65, IEEE, 2015.

[13] ONF, “OpenFlow Switch Specification,” tech. rep., Open Networking Foundation, October 2013.

[14] P. Rygielski, M. Seliuchenko, S. Kounev, and M. Klymash, “Performance Analysis of SDN Switches with Hardware and Software Flow Tables,”

[15] F. Benamrane, M. B. Mamoun, and R. Benaini, "Performances of OpenFlow-Based Software-Defined Networks: An overview.," *JNW*, vol. 10, no. 6, pp. 329–337, 2015.

[16] A. Ost, *Performance of communication systems: A model-based approach with matrix-geometric methods*. Springer Science & Business Media, 2013.

[17] M. N. Sadiku and S. M. Musa, *Performance analysis of computer networks*. Springer, 2013.

[18] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive software-defined network (TSSDN) for real-time applications," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pp. 193–202, ACM, 2016.

[19] M. Haiyan, Y. Jinyao, P. Georgopoulos, and B. Plattner, "Towards SDN based queuing delay estimation," *China Communications*, vol. 13, no. 3, pp. 27–36, 2016.

[20] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, "End-to-End Network Delay Guarantees for Real-Time Systems using SDN," in *IEEE Conference Real-Time Systems Symposium (RTSS)(Accepted)*, 2017.

[21] C. Olariu, M. Zuber, and C. Thorpe, "Delay-based priority queueing for VoIP over Software Defined Networks," in *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pp. 652–655, IEEE, 2017.

[22] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *Queue*, vol. 11, no. 12, p. 20, 2013.

[23] "SDN Architecture: Technical Report ONF TR-502." https://www.opennetworking.org/sdn-resources/sdn-definition.

[24] "Software-Defined Networking (SDN) Definition." `https://www.opennetworking.org/sdn-resources/sdn-definition`.

[25] 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, 2008.

[26] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework." RFC 3746 (Informational), April 2004.

[27] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," tech. rep., 2005.

[28] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (NETCONF)," tech. rep., 2011.

[29] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, "The locator/ID separation protocol (LISP)," tech. rep., 2013.

[30] OpenFlow Community, "OpenFlow Switching Reference System," 2009.

[31] "Floodlight REST API." `https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Floodlight+REST+API`.

[32] "OpenStackRESTAPI." `https://wiki.openstack.org/wiki/OpenStackRESTAPI`.

[33] V. Core, "July 2014," *URL http://www. vyatta. org*.

[34] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: session initiation protocol," tech. rep., 2002.

[35] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, “SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains.” Internet Draft, June 2012.

[36] P. Lin, J. Bi, and Y. Wang, “East-west bridge for SDN network peering,” in *Frontiers in Internet Technologies*, pp. 170–181, Springer, 2013.

[37] S. Hassas Yeganeh and Y. Ganjali, “Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN ’12, (New York, NY, USA), pp. 19–24, ACM, 2012.

[38] Q. Zhang, J. Liu, and G. Zhao, “Towards 5G Enabled Tactile Robotic Telesurgery,” *arXiv preprint arXiv:1803.03586*, 2018.

[39] “Open vSwitch.” `http://openvswitch.org/`.

[40] “Pantou: OpenFlow 1.3 for OpenWRT.” `https://github.com/CPqD/ofsoftswitch13/wiki/OpenFlow-1.3-for-OpenWRT`.

[41] “ofsoftswitch13.” `https://github.com/CPqD/ofsoftswitch13`.

[42] “Indigo: Open Source Openflow Switches.” `https://floodlight.atlassian.net/wiki/spaces/HOME/overview`.

[43] D. Y. Huang, K. Yocum, and A. C. Snoeren, “High-fidelity switch models for software-defined network emulation,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 43–48, ACM, 2013.

[44] M. Berman, J. S. Chase, L. H. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, “GENI: A federated testbed for innovative network experiments.,” *Computer Networks*, vol. 61, pp. 5–23, 2014.

[45] H. Harai, "AKARI architecture design for new generation network," in *Summer Topical Meeting, 2009. LEOSST '09. IEEE/LEOS*, pp. 155–156, IEEE, 2009.

[46] V. Kotronis, D. Schatzmann, and B. Ager, "On bringing private traffic into public SDN testbeds," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, (New York, NY, USA), pp. 155–156, ACM, 2013.

[47] Y. Kanaumi, S.-i. Saito, E. Kawai, S. Ishii, K. Kobayashi, and S. Shimojo, "RISE: A wide-area hybrid OpenFlow network testbed," *IEICE transactions on communications*, vol. 96, no. 1, pp. 108–118, 2013.

[48] M. Campanella and F. Farina, "The FEDERICA infrastructure and experience.," *Computer Networks*, vol. 61, pp. 176–183, 2014.

[49] A. Köpsel and H. Woesner, *OFELIA – Pan-European Test Facility for OpenFlow Experimentation*. Springer, 2011.

[50] T. Huang, F. R. Yu, C. Zhang, J. Liu, J. Zhang, and Y. Liu, "A survey on large-scale software defined networking (SDN) testbeds: Approaches and challenges," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 891–917, 2016.

[51] J. Kim, B. Cha, J. Kim, N. L. Kim, G. Noh, Y. Jang, H. G. An, H. Park, J. Hong, D. Jang, *et al.*, "OF@ TEIN: An OpenFlow-enabled SDN testbed over international SmartX Rack sites," *Proceedings of the Asia-Pacific Advanced Network*, vol. 36, pp. 17–22, 2013.

[52] P. Biondi, "Scapy: explore the net with new eyes," *Technical report, EADS Corporate Research Center*, 2005.

[53] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[54] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges," *Computer Networks*, vol. 72, pp. 74–98, 2014.

[55] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[56] G. N. Nde and R. Khondoker, "SDN testing and debugging tools: A survey," in *Informatics, Electronics and Vision (ICIEV), 2016 5th International Conference on*, pp. 631–635, IEEE, 2016.

[57] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *Global Communications Conference (GLOBECOM), 2013 IEEE*, pp. 1397–1402, IEEE, 2013.

[58] J.-Y. L. Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, vol. 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[59] C.-H. Ng and S. Boon-Hee, *Queueing modelling fundamentals: With applications in communication networks*. John Wiley & Sons, 2008.

[60] Z. Bozakov and A. Rizk, "Taming SDN controllers in heterogeneous hardware environments," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pp. 50–55, IEEE, 2013.

[61] S. Azodolmolky, P. Wieder, and R. Yahyapour, "Performance evaluation of a scalable software-defined networking deployment," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pp. 68–74, IEEE, 2013.

[62] A. K. Koohanestani, A. G. Osgouei, H. Saidi, and A. Fanian, "An analytical model for delay bound of OpenFlow based SDN using net-

work calculus," *Journal of Network and Computer Applications*, vol. 96, pp. 31–38, 2017.

[63] J. Huang, L. Xu, Q. Duan, C.-c. Xing, J. Luo, and S. Yu, "Modeling and performance analysis for multimedia data flows scheduling in software defined networks," *Journal of Network and Computer Applications*, vol. 83, pp. 89–100, 2017.

[64] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, "Quantitative System Performance Computer System Analysis Using Queueing Network Models.," in *Int. CMG Conference* (G. W. Dodson, H. P. Artis, D. R. Deese, B. Domanski, S. Finehirsh, J. Gaffney, and A. von Mayrhauser, eds.), pp. 468–470, Computer Measurement Group, 1983.

[65] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain," *The Annals of Mathematical Statistics*, pp. 338–354, 1953.

[66] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," in *Proceedings of the 23rd international teletraffic congress*, pp. 1–7, International Teletraffic Congress, 2011.

[67] K. Mahmood, A. Chilwan, O. N. Østerbø, and M. Jarschel, "On The Modeling of OpenFlow-based SDNs: The Single Node Case," pp. 207–214, Nov. 2014.

[68] K. Mahmood, A. Chilwan, O. N. Østerbø, and M. Jarschel, "Modelling of OpenFlow-based software-defined networks: the multiple node case," *Networks, IET*, vol. 4, no. 5, pp. 278–284, 2015.

[69] T.-C. Yen and C.-S. Su, "An SDN-based cloud computing architecture and its mathematical model," in *Information Science, Electron-*

ics and Electrical Engineering (ISEEE), 2014 International Conference on, vol. 3, pp. 1728–1731, IEEE, 2014.

[70] W. Miao, G. Min, Y. Wu, H. Wang, and J. Hu, "Performance modelling and analysis of software-defined networking under bursty multimedia traffic," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 12, no. 5s, p. 77, 2016.

[71] D. Singh, B. Ng, Y.-C. Lai, Y.-D. Lin, and W. K. Seah, "Modelling Software-Defined Networking: Switch Design with Finite Buffer and Priority Queueing," in *Local Computer Networks (LCN), 2017 IEEE 42nd Conference on*, pp. 567–570, IEEE, 2017.

[72] Z. Shang and K. Wolter, "Delay evaluation of openflow network based on queueing model," *arXiv preprint arXiv:1608.06491*, 2016.

[73] B. Xiong, K. Yang, J. Zhao, W. Li, and K. Li, "Performance evaluation of OpenFlow-based software-defined networks based on queueing model," *Computer Networks*, vol. 102, pp. 172–185, 2016.

[74] S. Muhizi, G. Shamshin, A. Muthanna, R. Kirichek, A. Vladyko, and A. Koucheryavy, "Analysis and performance evaluation of SDN queue model," in *International Conference on Wired/Wireless Internet Communication*, pp. 26–37, Springer, 2017.

[75] J. Ansell, W. K. Seah, B. Ng, and S. Marshall, "Making queueing theory more palatable to SDN/OpenFlow-based network practitioners," in *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pp. 1119–1124, IEEE, 2016.

[76] U. Javed, A. Iqbal, S. Saleh, S. A. Haider, and M. U. Ilyas, "A stochastic model for transit latency in OpenFlow SDNs," *Computer Networks*, vol. 113, pp. 218–229, 2017.

[77] K. Sood, S. Yu, and Y. Xiang, "Performance analysis of software-defined network switch using $M/Geo/1$ model," *IEEE Communications Letters*, vol. 20, no. 12, pp. 2522–2525, 2016.

[78] Y.-C. Lai, A. Ali, M. M. Hassan, M. S. Hossain, and Y.-D. Lin, "Performance Modeling and Analysis of TCP Connections over Software Defined Networks," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pp. 1–6, IEEE, 2017.

[79] A. Fahmin, Y.-C. Lai, M. S. Hossain, and Y.-D. Lin, "Performance modeling and comparison of NFV integrated with SDN: Under or aside?," *Journal of Network and Computer Applications*, vol. 113, pp. 119–129, 2018.

[80] G. Wang, J. Li, and X. Chang, "Modeling and performance analysis of the multiple controllers' approach in software defined networking," in *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*, pp. 73–74, IEEE, 2015.

[81] L. Abeni, D. Fontanelli, and L. Palopoli, "Application of the Quasi-Birth-Death Processes techniques to probablistic guarantees of soft realtime systems scheduled by resource reservations," *DISI-Universitá di Trento, Tech. Rep*, 2015.

[82] R. Chakka, "Performance and reliability modelling of computing systems using spectral expansion," 1995.

[83] H. Baumann and W. Sandmann, "Numerical solution of level dependent quasi-birth-and-death processes," *Procedia Computer Science*, vol. 1, no. 1, pp. 1561–1569, 2010.

[84] S. Hautphenne, K. Leibnitz, and M.-A. Remiche, "Modeling of P2P file sharing with a level-dependent QBD process," in *Advances in Queueing Theory and Network Applications*, pp. 247–263, Springer, 2009.

[85] T. Dayar, W. Sandmann, D. Spieler, and V. Wolf, "Infinite level-dependent QBD processes and matrix-analytic solutions for stochastic chemical kinetics," *Advances in Applied Probability*, vol. 43, no. 4, pp. 1005–1026, 2011.

[86] A. J. Motyer, *Quasi-birth-and-death Processes with an Infinite Phase Space*. University of Melbourne, Department of Mathematics and Statistics, 2011.

[87] G. Latouche and V. Ramaswami, *Introduction to matrix analytic methods in stochastic modeling*, vol. 5. Siam, 1999.

[88] J. P. Kharoufeh, "Level-Dependent Quasi-Birth-and-Death Processes," *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[89] M. F. Neuts, *Matrix-geometric solutions in stochastic models - an algorithmic approach.* Dover Publications, 1994.

[90] G. Latouche and V. Ramaswami, "A logarithmic reduction algorithm for quasi-birth-death processes," *Journal of Applied Probability*, pp. 650–674, 1993.

[91] D. Bini and B. Meini, "On the solution of a nonlinear matrix equation arising in queueing problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 906–926, 1996.

[92] W. J. Stewart, *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton University Press, 2009.

[93] N. Hegde and E. Altman, "Capacity of multiservice WCDMA Networks with variable GoS," in *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, vol. 2, pp. 1402–1407, IEEE, 2003.

[94] L. Bright and P. G. Taylor, "Calculating the equilibrium distribution in level dependent quasi-birth-and-death processes," *Stochastic Models*, vol. 11, no. 3, pp. 497–525, 1995.

[95] T. Hanschke, "A matrix continued fraction algorithm for the multiserver repeated order queue," *Mathematical and Computer Modelling*, vol. 30, no. 3-4, pp. 159–170, 1999.

[96] P. J. Burke, "The Output of a Queuing System," *Operations Research*, vol. 4, no. 6, pp. 699–704, 1956.

[97] A. Busic, B. Gaujal, and F. Perronnin, "Perfect Sampling of Networks with Finite and Infinite Capacity Queues.," in *ASMTA* (K. Al-Begain, D. Fiems, and J.-M. Vincent, eds.), vol. 7314 of *Lecture Notes in Computer Science*, pp. 136–149, Springer, 2012.

[98] S. Ogasawara and Y. Takahashi, "Performance analysis of traffic classification in an OpenFlow switch," in *Cloudification of the Internet of Things (CIoT)*, pp. 1–6, IEEE, 2016.

[99] "PicOS Support for OpenFlow 1.3." https://docs.pica8.com/display/PICOS2111cg/PicOS+Support+for+OpenFlow+1.3.

[100] "Cisco Plug-in for OpenFlow Configuration." https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3850/software/release/37e/b-openflow-37e-3850-and-3650/b-openflow-37e-3850_chapter_01.html.

[101] "HPE Switch Software OpenFlow v1.3 Administrator Guide." http://h22208.www2.hpe.com/eginfolib/networking/docs/switches/K-KA-KB/16-01/5200-0146OFAG/webhelp/content/index.html.

[102] "OpenFlow Support on Juniper Network Devices." https://www.juniper.net/documentation/en_US/release-independent/junos/topics/reference/general/junos-sdn-openflow-supported-platforms.html.

[103] "Arista EOS OpenFlow." https://www.arista.com/en/um-eos/eos-openflow.

[104] "ExtremeXOS OpenFlow User Guide." https://documentation.extremenetworks.com/openflow/exos_all/openflow/openflow.shtml.

[105] K. Mizuyama, Y. Taenaka, and K. Tsukamoto, "Estimation based adaptable Flow Aggregation Method for reducing control traffic on Software Defined wireless Networks," in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, (Kona, HI, USA), pp. 363–368, 13-17 Mar 2017.

[106] X. S. Sun, A. Agarwal, and T. E. Ng, "Controlling race conditions in openflow to accelerate application verification and packet forwarding," *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 263–277, 2015.

[107] K. Nguyen, K. Ishizu, and F. Kojima, "An evolvable, scalable, and resilient control channel for software defined wireless access networks," *Computers & Electrical Engineering*, vol. 57, pp. 104–117, 2017.

[108] C. Hu, K. Hou, H. Li, R. Wang, P. Zheng, P. Zhang, and H. Wang in *IEEE 25th International Conference on Network Protocols (ICNP)*.

[109] S. Ogasawara and Y. Takahashi, "Performance analysis of traffic classification in an OpenFlow switch," in *Proceedings of the Inter-*

national Conference on Cloudification of the Internet of Things (CIoT), (Paris, France), pp. 1–6, 23-25 Nov 2016.

[110] A. Mondal, S. Misra, and I. Maity, "Buffer Size Evaluation of OpenFlow Systems in Software-Defined Networks," *IEEE Systems Journal*, pp. 1–8, 2018.

[111] R. Schassberger, "Review of Reversibility and stochastic networks," *Perform. Eval.*, vol. 1, no. 1, p. 90, 1981.

[112] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains*. Hoboken, New Jersey: John Wiley & Sons, Inc., second ed., 2006.

[113] "ISO/IEC/IEEE International Standard for Ethernet," *ISO/IEC/IEEE 8802-3:2014(E)*, pp. 1–3754, April 2014.

[114] S. C. Liew, "Performance of various input-buffered and output-buffered ATM switch design principles under bursty traffic: simulation study," *IEEE Transactions on Communications*, vol. 42, pp. 1371–1379, Feb 1994.

[115] R. J. Simcoe and T.-B. Pei, "Perspectives on ATM Switch Architecture and the Influence of Traffic Pattern Assumptions on Switch Design," *SIGCOMM Comput. Commun. Rev.*, vol. 25, pp. 93–105, Apr 1995.

[116] R. Serfozo, *Basics of applied stochastic processes*. Springer Science & Business Media, 2009.

[117] Y. L. D. Y. Burman, J. P. Lehoczky, "Insensitivity of Blocking Probabilities in a Circuit-Switching Network," *Journal of Applied Probability*, vol. 21, no. 4, pp. 850–859, 1984.

[118] P. E. Heegaard, "Evolution of traffic patterns in telecommunication systems," in *Communications and Networking in China, 2007. CHINACOM'07. Second International Conference on*, pp. 28–32, IEEE, 2007.

[119] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," in *Proceedings of the 23rd international teletraffic congress*, pp. 1–7, International Teletraffic Congress, 2011.

[120] L. Zhao, J. Hua, X. Ge, and S. Zhong, "Traffic engineering in hierarchical SDN control plane," in *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*, pp. 189–194, IEEE, 2015.

[121] H. H. Kurmann and H. M. Kurmann, *On the Emulation of Impairments in ATM-networks*. No. 24, vdf Hochschulverlag AG, 1997.

[122] M. Appelman and M. de Boer, "Performance analysis of OpenFlow hardware," *University of Amsterdam, Tech. Rep*, pp. 2011–2012, 2012.

[123] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Passive and Active Measurement*, pp. 347–359, Springer, 2015.

[124] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: Enable flexible multi-table processing on legacy hardware," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 85–90, ACM, 2013.

[125] E. Gelenbe and I. Mitrani, *Analysis and synthesis of computer systems*, vol. 4. World Scientific, 2010.

[126] D. Singh, B. Ng, Y.-C. Lai, Y.-D. Lin, and W. K. Seah, "Modelling Software-Defined Networking: Software and hardware switches," *Journal of Network and Computer Applications*, 2018.

Appendix A

Logarithmic Reduction (LR) Method

The pseudocode to compute matrix G using LR method is shown in Algorithm 1 [92].

Algorithm 1: Logarithmic Reduction Method to compute matrix G.

```
Initialize:
$s = 0$;
$L = -A_1^{-1}A_2$;      ▷ Matrix $L$ describes transitions to lower level
$H = -A_1^{-1}A_0$;      ▷ Matrix $H$ describes transitions to upper level
$G = L$;
$T = H$;
while ($\|e - Ge\|_\infty > \epsilon$) do
    $s = s + 1$;
    $D = I - HL - LH$;               ▷ $I$ is identity matrix
    $L = D^{-1}L^2$;
    $H = D^{-1}H^2$;
    $G = G + TL$;
    $T = TH$;
```

The matrix R can be computed from the relationship between matrices R and G shown in [90],

$$R = -A_0(A_1 + A_0G)^{-1}. \tag{A.1}$$

Using matrix R and boundary equations of $\pi Q = 0$, the initial stationary distribution probabilities π_0 and π_1 can be computed. The boundary equations of $\pi Q = 0$ are shown as

$$\begin{aligned} \pi_0 B_1 + \pi_1 A_2 &= 0, \\ \pi_0 A_0 + \pi_1 A_1 + \pi_2 A_2 &= 0. \end{aligned} \tag{A.2}$$

Rewriting Eq. A.2 into matrix form after substituting $\pi_2 = \pi_1 R$, we get

$$(\pi_0, \pi_1) \begin{pmatrix} B_1 & A_0 \\ A_2 & A_1 + RA_2 \end{pmatrix} = 0. \tag{A.3}$$

The remaining stationary distribution probabilities for homogeneous QBD process can be computed using the general solution for π i.e. $\pi_i = \pi_1 R$ or $\pi_i = \pi_{i-1} R$, $i \geq 2$.

Appendix B

Matrix Continued Fraction Algorithm (MCF)

The computation of stationary distribution using MCF is shown in Algorithm 2.

Algorithm 2: A Matrix Continued Fraction Algorithm (MCF).

```
if $Q$ is infinite then
    Truncate infinite number of levels to finite large number $K$;
else
    The maximum number of levels of $Q$ is assigned to $K$;
$R_K = 0$;                          ▷ Set $R_K$ as zero matrix.
$R_{K-1} = -A_0^{(K-1)}(A_1^K)^{-1}$;          ▷ Compute $R_{K-1}$.
for $i = K-1, K-2, ...1$ do
    $R_{i-1} = -A_0^{i-1}(A_1^i + R_i A_2^{i+1})^{-1}$;     ▷ Compute remaining rate
     matrices.
$\pi_0(A_1^{(0)} + R_0 A_2^{(1)}) = 0$;             ▷ Estimate $\pi_0$.
for $i = 0, 1, ..., K-1$ do
    $\pi_{i+1} = \pi_i R_i$;        ▷ Compute remaining stationary distribution
     probabilities.
Normalize $\pi$.
```

Appendix C

Monte Carlo Simulation for Model SE

To work with events, a class "Event" is declared with a pointer object "eve" as shown below:

```
class Event{
    public:
        double time;
        double rate;
}*eve;
```

For model SE, there are three events i.e., n_event = 3 which are initialised with the transition rates of model SE as shown below:

- $eve[0].rate = \lambda_1$,
- $eve[1].rate = \mu_{sp}$, and
- $eve[2].rate = \mu_c$,

where λ_1 is the external arrival rate at the CPU, μ_{sp} is the CPU's processing rate, and μ_c is the controller's processing rate.

The processing times for each of these events can be computed using the function "edis" which is defined as,

```
double edis(double rate){
    return -1/rate*log(1-genrand_real1());
}
```

where "genrand_real1()" is a "MT.h" function which generates a random number on [0,1] real interval.

There are three processes related to three events of model SE. These processes are denoted by the variable "$flag$" $\in \{0, ..., n\_event - 1\}$ such that,

- $flag = 0$ represents the process for external packets arriving at the CPU,
- $flag = 1$ represents the process for a packet processing by the CPU, and
- $flag = 2$ represents the process for a packet processing by the controller.

Similarly, the number of packets for each queue is given by an array "$N[n\_event]$". As seen in Fig. 3.1, there are two queues for model SE for which the array N is given as,

- $N[0] =$ Number of external packets arriving at the switch,
- $N[1] =$ Number of packets at the controller forwarded by the switch, and
- $N[2] =$ Number of packets feedback by the controller to the switch.

Using these parameters, the pseudo code for Monte Carlo simulation of model SE is shown in Algorithm 3. The list of variables used in pseudo code for model SE and remaining models are:

- "$clock$" stores the clock time.
- "$ptime$" holds the value of processing time for the next event.

- “AT” and “DT” are arrays to store the arrival time and departure time of the n^{th} packet, respectively.
- “an” and “dn” are the identifiers for the n^{th} packet arriving at the switch and departing from the switch, respectively.
- “inq” is an array to store the number of packets sent to the controller for inquiry.
- “$inqn$” is the identifier for the packet sent to the controller for inquiry.
- “NSD” stores the total number of packets at the switch.
- “$loss$” stores the number of packet dropped or lost at the switch.
- “odr” is the character array to store the values ‘0’ and ‘1’ to distinguish packets arriving at the switch. ‘0’ represents the external packet arriving at the switch and ‘1’ represents the packet feedback by the controller to the switch.
- “NOP” is the maximum number of packets entering into the system.
- “K” is the finite queue capacity for the shared buffer in the CPU.
- “β” is the table miss probability.
- “nos” represents the maximum number of simulation.
- “$NPTD$” is an array to store the packet transfer delay for each simulation.
- “NPL” is an array to store the packet loss for each simulation.
- “PTD” is the variable to store an average packet transfer delay.
- “PL” is the variable to store an average packet loss rate.

Algorithm 3: Pseudo code for model SE.

```
for $count = 0, ..., nos - 1$ do
    Initialise:
            $inqn = NSD = an = dn = loss = clock = 0$;
            $N[0] = N[1] = N[2] = 0$;
            $eve[0].time = edis(eve[0].rate)$;    ▷ model is activated with
    external packets arriving at the CPU.
    do
        Selection of the next event as shown in Algorithm 4;
        $clock$ += $ptime$;
        switch flag do
            case 0 do
                Pseudo code for the flag = 0 case as shown in Algorithm 5;
            case 1 do
                Pseudo code for the flag = 1 case as shown in Algorithm 6;
            case 2 do
                Pseudo code for the flag = 2 case as shown in Algorithm 7;
    while ($dn < NOP$ && $inq[0] < NOP$)
    $NPTD[count] = 0$;
    for $i = 0, ..., dn - 1$ do
        if $DT[i] \neq 0$ then
            $NPTD[count]$+=$DT[i] - AT[i]$;
    $NPL[count] = loss/dn$;
    $NPTD[count]$ /= $dn$;
$PL = PTD = 0$;
for $count = 0, ..., nos - 1$ do
    $PTD$+=$NPTD[count]$;
    $PL$+=$NPL[count]$;
$PTD/$=$nos$;                  ▷ average packet transfer delay
$PL/$=$nos$;                   ▷ average packet loss rate
```

Algorithm 4: Pseudo code for the selection of next event in model SE.

```
Initialise:
        $ptime = \inf$;
if ($N[0] == 0$ && $N[2] == 0$) then
    $eve[1].time = \inf$;              ▷ if the CPU has no packets.
if ($N[1] == 0$) then
    $eve[2].time = \inf$;              ▷ if the controller has no packets.
for $i = 0, 1, ..., n\_event - 1$ do
    if ($eve[i].time == 0$) then
        $eve[i].time = edis(eve[i].rate)$;     ▷ Update the event's processing
          time if equal to zero.
    if ($ptime > eve[i].time$) then
        $ptime = eve[i].time$;
        $flag = i$;     ▷ Select the event with smallest processing time.
for $i = 0, 1, ..., n\_event - 1$ do
    $eve[i].time = eve[i].time - ptime$; ▷ Update the processing time for all
      events by subtracting the smallest processing time.
```

Algorithm 5: Pseudo code for the flag equal to '0' in model SE.

```
if ($N[0] + N[2] < K$) then
    $AT[an] = clock$;▷ assigning the clock value to "$an$" packets arriving at
      the switch.
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $odr[NSD]$ = '0';
    $odr[NSD$+1] = '\0';
    $NSD$++; $an$++;
    $N[0]$++;                  ▷ one packet arrives at the CPU.
else
    $loss$++;
```

Algorithm 6: Pseudo code for the flag equal to '1' in model SE.

```
if (genrand_real1() $< \beta$) then
    if ($odr[0]$ =='0') then
        $N[0]$- -;   ▷ one packet forwarded from the CPU to the controller.
        $inq[inqn] = dn$;
        $inqn$++;
        if ($N[1] == 0$) then
            $eve[2].time = edis(eve[2].rate)$;
        $N[1]$++;  ▷ one packet forwarded from the CPU to the controller.
        $dn$++;
    else if ($odr[0]$ == '1') then
        $N[2]$- -;      ▷ one packet serviced by the controller to the CPU.
        $temp = inq[0]$;
        for $k = 0, ..., inqn - 2$ do
            $inq[k] = inq[k+1]$;
        $inq[k] = temp$;
        if ($N[1] == 0$) then
            $eve[2].time = edis(eve[2].rate)$;
        $N[1]$++;
else
    if ($odr[0]$ =='0') then
        $N[0]$- -;      ▷ one packet departs from the CPU to out of the system
          (SE).
        $DT[dn] = clock$;
        $dn$ - -;
    else if ($odr[0]$ =='1') then
        $N[2]$- -; ▷ one packet serviced by the controller to the CPU departs
          out of the system (SE).
        $DT[inq[0]] = clock$;
        for $j = 0, ..., inqn - 1$ do
            $inq[j] = inq[j+1]$;
        $inqn$ - -;
$NSD$ - -;
for $j = 0, ..., NSD - 1$ do
    $odr[j] = odr[j+1]$;
$odr[NSD]$ = '\0';
```

Algorithm 7: Pseudo code for the flag equal to '2' in model SE.

```
$N[1]$- -;            ▷ one packet serviced by the controller to the CPU.
if ($N[0] + N[2] < K$) then
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[2]$++;   ▷ one packet serviced by the controller arrives at the CPU.
    $odr[NSD]$ = '1';
    $odr[NSD$+1$]$ = '\0';
    $NSD$++;
else
    $loss$++;
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$- -;
```

Appendix D

Monte Carlo Simulation for Model SPE

Similar to model SE, there are three events for model SPE i.e., $n\_event = 3$ which are initialised with the transition rates of model SPE as shown below:

- $eve[0].rate = \lambda_1$,
- $eve[1].rate = \mu_{sp}$, and
- $eve[2].rate = \mu_c$,

where λ_1 is the external arrival rate at the Class ES of the CPU, μ_{sp} is the CPU's processing rate, and μ_c is the controller's processing rate.

The processing times for each of these events can be computed using the function "edis" as discussed in Appendix C.

There are three processes related to three events of model SPE. These processes are denoted by the variable "$flag$" $\in \{0, ..., n\_event - 1\}$ such that,

- $flag = 0$ represents the process for external packets arriving at the Class ES of the CPU,

- $flag = 1$ represents the process for a packet processing by the CPU, and
- $flag = 2$ represents the process for a packet processing by the controller.

As seen in Fig. 3.2, there are three queues for model SPE such that the array "N" is given as,

- $N[0]$ = Number of external packets arriving at the Class ES of the CPU,
- $N[1]$ = Number of packets at the controller forwarded by the CPU, and
- $N[2]$ = Number of packets feedback by the controller to the Class CS of the CPU.

The finite queue capacity of the CPU (denoted by K) for model SPE is the sum of queue capacities of the Class CS (denoted by K_1) and Class ES (denoted by K_2) of the CPU, i.e. $K = K_1 + K_2$. Using these parameters, the pseudo code for Monte Carlo simulation of model SPE is shown in Algorithm 8.

Algorithm 8: Pseudo code for model SPE.

```
for $count = 0, ..., nos - 1$ do
    Initialise:
            $inqn = an = dn = loss = clock = 0$;
            $N[0] = N[1] = N[2] = 0$;
            $eve[0].time = edis(eve[0].rate)$;    ▷ model is activated with
    external packets arriving at the CPU.
    do
        Selection of the next event as shown in Algorithm 9;
        $clock$ += $ptime$;
        switch flag do
            case 0 do
                Pseudo code for the flag = 0 case as shown in Algorithm 10;
            case 1 do
                Pseudo code for the flag = 1 case as shown in Algorithm 11;
            case 2 do
                Pseudo code for the flag = 2 case as shown in Algorithm 12;
    while ($dn < NOP$ && $inq[0] < NOP$)
    $NPTD[count] = 0$;
    for $i = 0, ..., dn - 1$ do
        if $DT[i] \neq 0$ then
            $NPTD[count]$+=$DT[i] - AT[i]$;
    $NPL[count] = loss/dn$;
    $NPTD[count]$ /= $dn$;
$PL = PTD = 0$;
for $count = 0, ..., nos - 1$ do
    $PTD$+=$NPTD[count]$;
    $PL$+=$NPL[count]$;
$PTD/$=$nos$;                ▷ average packet transfer delay
$PL/$=$nos$;                 ▷ average packet loss rate
```

Algorithm 9: Pseudo code for the selection of next event in model SPE.

```
Initialise:
        $ptime = \mathrm{inf}$;
if ($N[0] == 0$ && $N[2] == 0$) then
    $eve[1].time = \mathrm{inf}$;            ▷ if the CPU has no packets.
if ($N[1] == 0$) then
    $eve[2].time = \mathrm{inf}$;            ▷ if the controller has no packets.
for $i = 0, 1, ..., n\_event - 1$ do
    if ($eve[i].time == 0$) then
        $eve[i].time = edis(eve[i].rate)$;    ▷ Update the event's processing
          time if equal to zero.
    if ($ptime > eve[i].time$) then
        $ptime = eve[i].time$;
        $flag = i$;        ▷ Select the event with smallest processing time.
for $i = 0, 1, ..., n\_event - 1$ do
    $eve[i].time = eve[i].time - ptime$; ▷ Update the processing time for all
      events by subtracting the smallest processing time.
```

Algorithm 10: Pseudo code for the flag equal to '0' in model SPE.

```
if ($N[0] < K_2$) then
    $AT[an] = clock$;▷ assigning the clock value to "$an$" packets arriving at
      the switch.
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[0]$++;                ▷ one packet arrives at the Class ES.
    $an$++;
else
    $loss$++;
```

Algorithm 11: Pseudo code for the flag equal to '1' in model SPE.

```
if ($N[2] > 0$) then
    $N[2]$- -;    ▷ one packet departs from the Class CS out of the system (SPE).
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$ - -;
else
    $N[0]$- -;    ▷ one packet departs from the Class ES.
    if (genrand_real1() < $\beta$) then
        $inq[inqn] = dn$;
        $inqn$++;
        if ($N[1] == 0$) then
            $eve[2].time = edis(eve[2].rate)$;
        $N[1]$++;    ▷ one packet forwarded from the Class ES to the controller.
        $dn$++;
    else
        $DT[dn] = clock$;
        $dn$++;
```

Algorithm 12: Pseudo code for the flag equal to '2' in model SPE.

```
$N[1]$- -;          ▷ one packet serviced by the controller to the Class CS.
if ($N[2] < K_1$) then
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[2]$++;▷ one packet serviced by the controller arrives at the Class CS.
else
    $loss$++;
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$- -;
```

Appendix E

Monte Carlo Simulation for Model SPI

Similar to model SE and model SPE, there are three events for model SPI i.e., n_event = 3 which are initialised with the transition rates of model SPI as shown below:

- $eve[0].rate = \lambda_1$,
- $eve[1].rate = \mu_{sp}$, and
- $eve[2].rate = \mu_c$,

where λ_1 is the external arrival rate at the Class ES of the CPU, μ_{sp} is the CPU's processing rate, and μ_c is the controller's processing rate.

The processing times for each of these events can be computed using the function "edis" as discussed in Appendix C.

There are three processes related to three events of model SPI. These processes are denoted by the variable "$flag$" $\in \{0, ..., n\_event - 1\}$ such that,

- $flag = 0$ represents the process for external packets arriving at the Class ES of the CPU,

- $flag = 1$ represents the process for a packet processing by the CPU, and
- $flag = 2$ represents the process for a packet processing by the controller.

As seen in Fig. 4.1, there are four queues for model SPI such that the array "N" is given as,

- $N[0]$ = Number of external packets arriving at the Class ES of the CPU,
- $N[1]$ = Number of packets at the controller forwarded by the CPU, and
- $N[2]$ = Number of packets feedback by the controller to the Class CS of the CPU.
- $N[3]$ = Number of packets temporarily buffered at the internal buffer of the CPU.

Using these parameters, the pseudo code for Monte Carlo simulation of model SPI is shown in Algorithm 13.

Algorithm 13: Pseudo code for model SPI.

```
for $count = 0, ..., nos - 1$ do
    Initialise:
            $inqn = an = dn = loss = clock = 0$;
            $N[0] = N[1] = N[2] = N[3] = 0$;
            $eve[0].time = edis(eve[0].rate)$;    ▷ model is activated with
    external packets arriving at the CPU.
    do
        Selection of the next event as shown in Algorithm 14;
        $clock$ += $ptime$;
        switch flag do
            case 0 do
                Pseudo code for the flag = 0 case as shown in Algorithm 15;
            case 1 do
                Pseudo code for the flag = 1 case as shown in Algorithm 16;
            case 2 do
                Pseudo code for the flag = 2 case as shown in Algorithm 17;
    while ($dn < NOP$ && $inq[0] < NOP$)
    $NPTD[count] = 0$;
    for $i = 0, ..., dn - 1$ do
        if $DT[i] \neq 0$ then
            $NPTD[count]$+=$DT[i] - AT[i]$;
    $NPL[count] = loss/dn$;
    $NPTD[count]$ /= $dn$;
$PL = PTD = 0$;
for $count = 0, ..., nos - 1$ do
    $PTD$+=$NPTD[count]$;
    $PL$+=$NPL[count]$;
$PTD/$=$nos$;                ▷ average packet transfer delay
$PL/$=$nos$;                 ▷ average packet loss rate
```

Algorithm 14: Pseudo code for the selection of next event in model SPI.

```
Initialise:
        $ptime = \inf$;
if ($N[0] == 0$ && $N[2] == 0$) then
    $eve[1].time = \inf$;                ▷ if the CPU has no packets.
if ($N[1] == 0$) then
    $eve[2].time = \inf$;                ▷ if the controller has no packets.
for $i = 0, 1, ..., n\_event - 1$ do
    if ($eve[i].time == 0$) then
        $eve[i].time = edis(eve[i].rate)$;      ▷ Update the event's processing
         time if equal to zero.
    if ($ptime > eve[i].time$) then
        $ptime = eve[i].time$;
        $flag = i$;      ▷ Select the event with smallest processing time.
for $i = 0, 1, ..., n\_event - 1$ do
    $eve[i].time = eve[i].time - ptime$; ▷ Update the processing time for all
     events by subtracting the smallest processing time.
```

Algorithm 15: Pseudo code for the flag equal to '0' in model SPI.

```
if ($N[0] < K_2$) then
    $AT[an] = clock$;▷ assigning the clock value to "$an$" packets arriving at
     the CPU.
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[0]$++;                ▷ one packet arrives at the Class ES.
    $an$++;
else
    $loss$++;
```

Algorithm 16: Pseudo code for the flag equal to '1' in model SPI.

```
if ($N[2] > 0$) then
    $N[2]$- -;        ▷ one packet in Class CS is processed by the CPU.
    $N[3]$- -;        ▷ one data packet is extracted from the internal buffer.
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$ - -;
else
    $N[0]$- -;        ▷ one packet departs from the Class ES.
    if (genrand_real1() < $\beta$) then
        $inq[inqn] = dn$;
        $N[3]$++;     ▷ one data packet is temporarily buffered in the internal
          buffer.
        $inqn$++;
        if ($N[1] == 0$) then
            $eve[2].time = edis(eve[2].rate)$;
        $N[1]$++;     ▷ one packet with a part of a data packet encapsulated
          with a packet_in message is forwarded to the controller.
        $dn$++;
    else
        $DT[dn] = clock$;
        $dn$++;
```

Algorithm 17: Pseudo code for the flag equal to '2' in model SPI.

```
if ($N[2] < K_1$) then
    $N[1]$- -;     ▷ one packet serviced by the controller to the Class CS.
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[2]$++;▷ one packet serviced by the controller arrives at the Class CS.
else
    $loss$++;
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$- -;
```

Appendix F

Monte Carlo Simulation for Model HPE

In model HPE, there are four events i.e., n_event = 4 which are initialised with the transition rates of model HPE as shown below:

- $eve[0].rate = \lambda_1$,
- $eve[1].rate = \mu_{sp}$,
- $eve[2].rate = \mu_c$, and
- $eve[3].rate = \mu_{sh}$.

where λ_1 is the external arrival rate at the switch hardware, μ_{sp} is the CPU's processing rate, μ_c is the controller's processing rate, and μ_{sh} is the switch hardware's processing rate.

The processing times for each of these events can be computed using the function "edis" as discussed in Appendix C.

There are four processes related to four events of model HPE. These processes are denoted by the variable "$flag$" $\in \{0, ..., n\_event - 1\}$ such that,

- $flag = 0$ represents the process for external packets arriving at the switch hardware,

- $flag = 1$ represents the process for a packet processing by the CPU,
- $flag = 2$ represents the process for a packet processing by the controller, and
- $flag = 3$ represents the process for a packet processing by the switch hardware.

As seen in Fig. 5.2, there are four queues for model HPE such that the array "N" is given as,

- $N[0]$ = Number of packets forwarded by the switch hardware to the Class HP of the CPU,
- $N[1]$ = Number of packets at the controller forwarded by the CPU,
- $N[2]$ = Number of packets feedback by the controller to the Class CP of the CPU, and
- $N[3]$ = Number of external packets arriving at the switch hardware.

The finite queue capacities for the Class CS, Class ES, and the switch hardware for model HPE are denoted by K_1, K_2, and K_3, respectively. Using these parameters, the pseudo code for Monte Carlo simulation of model HPE is shown in Algorithm 18.

Algorithm 18: Pseudo code for model HPE.

```
for $count = 0, ..., nos - 1$ do
    Initialise:
            $inqn = an = dn = loss = clock = 0$;
            $N[0] = N[1] = N[2] = N[3] = 0$;
            $eve[3].time = edis(eve[3].rate)$;    ▷ model is activated with
    external packets arriving at the switch hardware.
    do
        Selection of the next event as shown in Algorithm 19;
        $clock$ += $ptime$;
        switch flag do
            case 0 do
                Pseudo code for the flag = 0 case as shown in Algorithm 20;
            case 1 do
                Pseudo code for the flag = 1 case as shown in Algorithm 21;
            case 2 do
                Pseudo code for the flag = 2 case as shown in Algorithm 22;
            case 3 do
                Pseudo code for the flag = 3 case as shown in Algorithm 23;
    while ($dn < NOP$ && $inq[0] < NOP$)
    $NPTD[count] = 0$;
    for $i = 0, ..., dn - 1$ do
        if $DT[i] \neq 0$ then
            $NPTD[count]$+=$DT[i] - AT[i]$;
    $NPL[count] = loss/dn$;
    $NPTD[count]$ /= $dn$;
$PL = PTD = 0$;
for $count = 0, ..., nos - 1$ do
    $PTD$+=$NPTD[count]$;
    $PL$+=$NPL[count]$;
$PTD/$=$nos$;                ▷ average packet transfer delay
$PL/$=$nos$;                 ▷ average packet loss rate
```

Algorithm 19: Pseudo code for the selection of next event in model HPE.

```
Initialise:
        $ptime = \inf$;
if ($N[0] == 0$ && $N[2] == 0$) then
    $eve[1].time = \inf$;          ▷ if the CPU has no packets.
if ($N[1] == 0$) then
    $eve[2].time = \inf$;          ▷ if the controller has no packets.
if ($N[3] == 0$) then
    $eve[3].time = \inf$;          ▷ if the switch hardware has no packets.
for $i = 0, 1, ..., n\_event - 1$ do
    if ($eve[i].time == 0$) then
        $eve[i].time = edis(eve[i].rate)$;     ▷ Update the event's processing
          time if equal to zero.
    if ($ptime > eve[i].time$) then
        $ptime = eve[i].time$;
        $flag = i$;      ▷ Select the event with smallest processing time.
for $i = 0, 1, ..., n\_event - 1$ do
    $eve[i].time = eve[i].time - ptime$; ▷ Update the processing time for all
      events by subtracting the smallest processing time.
```

Algorithm 20: Pseudo code for the flag equal to '0' in model HPE.

```
if ($N[3] < K_3$) then
    $AT[an] = clock$;▷ assigning the clock value to "$an$" packets arriving at
      the switch hardware.
    if ($N[3] == 0$) then
        $eve[3].time = edis(eve[3].rate)$;
    $N[3]$++;            ▷ one packet arrives at the switch hardware.
    $an$++;
else
    $loss$++;
```

Algorithm 21: Pseudo code for the flag equal to '1' in model HPE.

```
if ($N[2] > 0$) then
    $N[2]$- -;        ▷ one packet departs from the Class CP out of the system (HPE).
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j+1]$;
    $inqn$ - -;
else
    $N[0]$- -;                ▷ one packet departs from the Class HP.
    if ($N[1] == 0$) then
        $eve[2].time = edis(eve[2].rate)$;
    $N[1]$++;  ▷ one packet forwarded from the Class HP to the controller.
```

Algorithm 22: Pseudo code for the flag equal to '2' in model HPE.

```
$N[1]$- -;        ▷ one packet serviced by the controller to the Class CP.
if ($N[2] < K_1$) then
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[2]$++;▷ one packet serviced by the controller arrives at the Class CP.
else
    $loss$++;
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j+1]$;
    $inqn$- -;
```

Algorithm 23: Pseudo code for the flag equal to '3' in model HPE.

```
$N[3]$- -;              ▷ one packet departs from the switch hardware.
if (genrand_real1() $< \beta$) then
    $inq[inqn] = dn$;
    $inqn$++;
    $dn$++;
    if ($N[0] < K_2$) then
        if ($N[0] == 0$ && $N[2] == 0$) then
            $eve[1].time = edis(eve[1].rate)$;
        $N[0]$++;  ▷ one packet arrives at the Class HP for CPU processing.
    else
        $loss$++;
        $DT[inq[inqn - 1]] = clock$;
        $inqn$- -;
else
    $DT[dn] = clock$;      ▷ one packet departs from the switch hardware to
      out of the system (HPE).
    $dn$++;
```

Appendix G

Monte Carlo Simulation for Model HPI

Similar to model HPE, there are four events in model HPI i.e., n_event = 4 which are initialised with the transition rates of model HPI as shown below:

- $eve[0].rate = \lambda_1$,
- $eve[1].rate = \mu_{sp}$,
- $eve[2].rate = \mu_c$, and
- $eve[3].rate = \mu_{sh}$.

where λ_1 is the external arrival rate at the switch hardware, μ_{sp} is the CPU's processing rate, μ_c is the controller's processing rate, and μ_{sh} is the switch hardware's processing rate.

The processing times for each of these events can be computed using the function "edis" as discussed in Appendix C.

There are four processes related to four events of model HPI. These processes are denoted by the variable "$flag$" $\in \{0, ..., n\_event - 1\}$ such that,

- $flag = 0$ represents the process for external packets arriving at the switch hardware,

- $flag = 1$ represents the process for a packet processing by the CPU,
- $flag = 2$ represents the process for a packet processing by the controller, and
- $flag = 3$ represents the process for a packet processing by the switch hardware.

As seen in Fig. 5.3, there are five queues for model HPI such that the array "N" is given as,

- $N[0]$ = Number of packets forwarded by the switch hardware to the Class HP of the CPU,
- $N[1]$ = Number of packets at the controller forwarded by the CPU,
- $N[2]$ = Number of packets feedback by the controller to the Class CP of the CPU,
- $N[3]$ = Number of external packets arriving at the switch hardware, and
- $N[4]$ = Number of packets temporarily buffered at the internal buffer of the CPU.

Using these parameters, the pseudo code for Monte Carlo simulation of model HPI is shown in Algorithm 24.

Algorithm 24: Pseudo code for model HPI.

```
for $count = 0, ..., nos - 1$ do
    Initialise:
            $inqn = an = dn = loss = clock = 0$;
            $N[0] = N[1] = N[2] = N[3] = N[4] = 0$;
            $eve[3].time = edis(eve[3].rate)$;    ▷ model is activated with
    external packets arriving at the switch hardware.
    do
        Selection of the next event as shown in Algorithm 25;
        $clock$ += $ptime$;
        switch flag do
            case 0 do
                Pseudo code for the flag = 0 case as shown in Algorithm 26;
            case 1 do
                Pseudo code for the flag = 1 case as shown in Algorithm 27;
            case 2 do
                Pseudo code for the flag = 2 case as shown in Algorithm 28;
            case 3 do
                Pseudo code for the flag = 3 case as shown in Algorithm 29;
    while ($dn < NOP$ && $inq[0] < NOP$)
    $NPTD[count] = 0$;
    for $i = 0, ..., dn - 1$ do
        if $DT[i] \neq 0$ then
            $NPTD[count]$+=$DT[i] - AT[i]$;
    $NPL[count] = loss/dn$;
    $NPTD[count]$ /= $dn$;
$PL = PTD = 0$;
for $count = 0, ..., nos - 1$ do
    $PTD$+=$NPTD[count]$;
    $PL$+=$NPL[count]$;
$PTD/$=$nos$;                ▷ average packet transfer delay
$PL/$=$nos$;                 ▷ average packet loss rate
```

Algorithm 25: Pseudo code for the selection of next event in model HPI.

```
Initialise:
        $ptime = \text{inf}$;
if ($N[0] == 0$ && $N[2] == 0$) then
    $eve[1].time = \text{inf}$;        ▷ if the CPU has no packets.
if ($N[1] == 0$) then
    $eve[2].time = \text{inf}$;        ▷ if the controller has no packets.
if ($N[3] == 0$) then
    $eve[3].time = \text{inf}$;        ▷ if the switch hardware has no packets.
for $i = 0, 1, ..., n\_event - 1$ do
    if ($eve[i].time == 0$) then
        $eve[i].time = edis(eve[i].rate)$;    ▷ Update the event's processing
          time if equal to zero.
    if ($ptime > eve[i].time$) then
        $ptime = eve[i].time$;
        $flag = i$;    ▷ Select the event with smallest processing time.
for $i = 0, 1, ..., n\_event - 1$ do
    $eve[i].time = eve[i].time - ptime$; ▷ Update the processing time for all
      events by subtracting the smallest processing time.
```

Algorithm 26: Pseudo code for the flag equal to '0' in model HPI.

```
if ($N[3] < K_3$) then
    $AT[an] = clock$;▷ assigning the clock value to "$an$" packets arriving at
      the switch hardware.
    if ($N[3] == 0$) then
        $eve[3].time = edis(eve[3].rate)$;
    $N[3]$++;        ▷ one packet arrives at the switch hardware.
    $an$++;
else
    $loss$++;
```

Algorithm 27: Pseudo code for the flag equal to '1' in model HPI.

```
if ($N[2] > 0$) then
    $N[2]$- -;          ▷ one packet in Class CP is processed by the CPU.
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$ - -;
    $N[4]$- -;       ▷ one data packet is extracted from the internal buffer.
else
    $N[0]$- -;                  ▷ one packet departs from the Class HP.
    if ($N[1] == 0$) then
        $eve[2].time = edis(eve[2].rate)$;
    $N[1]$++;  ▷ one packet forwarded from the Class HP to the controller.
    $N[4]$++;         ▷ one data packet is temporarily buffered in the internal
      buffer.
```

Algorithm 28: Pseudo code for the flag equal to '2' in model HPI.

```
$N[1]$- -;         ▷ one packet serviced by the controller to the Class CP.
if ($N[2] < K_1$) then
    if ($N[0] == 0$ && $N[2] == 0$) then
        $eve[1].time = edis(eve[1].rate)$;
    $N[2]$++;▷ one packet serviced by the controller arrives at the Class CP.
else
    $loss$++;
    $DT[inq[0]] = clock$;
    for $j = 0, ..., inqn - 1$ do
        $inq[j] = inq[j + 1]$;
    $inqn$- -;
```

Algorithm 29: Pseudo code for the flag equal to '3' in model HPI.

```
$N[3]$- -;                ▷ one packet departs from the switch hardware.
if (genrand_real1() $< \beta$) then
    $inq[inqn] = dn$;
    $inqn$++;
    $dn$++;
    if ($N[0] < K_2$) then
        if ($N[0] == 0$ && $N[2] == 0$) then
            $eve[1].time = edis(eve[1].rate)$;
        $N[0]$++; ▷ one packet arrives at the Class HP for CPU processing.
    else
        $loss$++;
        $DT[inq[inqn - 1]] = clock$;
        $inqn$- -;
else
    $DT[dn] = clock$;      ▷ one packet departs from the switch hardware to
      out of the system (HPI).
    $dn$++;
```