國立清華大學電機資訊學院資訊工程研究所
碩士論文
Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University
Master Thesis

以Mininet/Open vSwitch為基礎開發提供詳盡資訊的OpenFlow模擬器
Turning Mininet/Open vSwitch into A Detailed OpenFlow Emulator

鄭伊君
Yi-Jun Cheng

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 104 年 10 月
October, 2015

國立清華大學
資訊工程研究所

碩士論文

以Mininet/Open vSwitch為基礎開發提供詳盡資訊的OpenFlow模擬器

鄭伊君 撰

104
10

中文摘要

軟體定義網路 (SDN) 是一個新興的網路架構，它開啓了網路可程式化的可能性。 對於正在發展中的許多軟體定義網路研究計劃而言，開發一個可以高度準確地模擬使用 OpenFlow 的軟體定義網路的模擬器是必須的，如此以來才能更正確的驗證並評估嶄新的研究想法及方向。 但是，目前的模擬器大多是只專注在數據平面的效能 (data plane performances) 或是只專注於以軟體實作 OpenFlow 交換機 (software-implemented switches)。 這使得我們想要去開發一個 OpenFlow 網路模擬器，可以兼具控制平面 (control plane) 以及數據平面 (data plane) 效能的正確性模擬，並且可以模擬現在市場上不同廠牌的 OpenFlow 交換機。 在這篇論文裡，我們在不同的交換機上測量了控制平面及數據平面的效能 (control plane and data plane performances) 並提出了提高模擬正確性的效能模型 (performance models)。我們也提出了自動化地量測交換機效能的方法。 在我們提出的模型中，有一些可以調整的參數，調整這些參數是為了讓模型可以模擬來自不同廠商，不同實作的交換機。 這些參數會從我們提出的自動化效能量測實驗的結果所獲得。 我們執行了實驗去驗證效能模型的正確性，且錯誤率大部分在30%以下。 再者，我們也將效能模型整合到現有的開源軟體計劃中的 OpenFlow 模擬器，Mininet/Open vSwitch (OvS)，並且進行實驗驗證效能的正確性。我們將實驗結果與尚未修改的 Mininet/OvS 和欲模擬的交換機的結果一起做比較，整合效能模型的結果比起 Mininet/OvS 更正確地模擬了交換機的效能。

Abstract

Software-Defined Networking (SDN) is an emerging network architecture that enables network programmability and efficient network management. Recent research activities on SDN make it important to develop an emulator that accurately emulates OpenFlow-enabled SDN networks in order to verify and evaluate the innovative research ideas. However, existing emulators and simulators focus on either data plane performance or software-implemented switches. This motivates us to develop an OpenFlow emulator that provides accurate emulation on both control plane and data plane performances of an OpenFlow network and supports diverse OpenFlow switch implementations. In this thesis, we conduct extensive measurement studies on control plane and data plane performances on several switches and propose performance models for accurate emulation. Automatic switch performance measurements are also derived. In our proposed models, we have configurable switch-dependent parameters that characterize different switch implementations. Those parameters are generated from our automatic switch performance measurements. We conduct experiments to validate our performance models, and the error rates are mostly under 30%. Moreover, we integrate our performance models with a popular open source OpenFlow emulator, Mininet/Open vSwitch (OvS) and evaluate the performance accuracy by comparing to the results of original Mininet/OvS and the emulated switch. Our results are far more close to the emulated switch than the original Mininet/OvS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In traditional networks, each switch in networks makes its own decisions on where to forward incoming packets, which is done in the control plane of switches. Each switch computes the routing table locally. The data plane of switches manages packet forwarding according to the routing decisions. Software-Defined Networking (SDN), a newly-emerging and developing network architecture, changes the way. Different from traditional networks, SDN decouples the control plane from switches and offloads it to a remote controller. Network administrators can then develop and deploy network applications on controllers so as to manage networks in a cost-effective way or provide novel services with less effort.

Since the control plane of switches is offloaded to a remote controller, messages exchanging between controllers and switches become inevitable. OpenFlow protocol [29], the communication standard between the controller and the switch, is thus established. There come two common paradigms for controllers to manage a SDN-based network: proactive and reactive approaches [16]. Controllers either proactively install flows in advance according to the traffic patterns of the network or reactively update flow table in response to the new flows in the data plane. Majority SDN-based solutions adopt reactive flow operation paradigm [40]. Since packets that do no match any flow in the switch flow table are *PacketIn* to the controller for forwarding decisions, the processing delay (at the controllers) and network delay (between the switch and controller) *increases* the latency of the first packet of each flow. Moreover, due to the centralized nature, SDN controllers are also vulnerable to staggering instantaneous workload. Both factors impose direct and dramatic negative impacts on interactivity of SDN networks, and thus the *control plane performance* of each switch is crucial to user experience in SDN networks.

Furthermore, OpenFlow operations are very flexible compared to those supported by the traditional L2/L3 switch Application-Specific Integrated Circuit (ASIC) chips. Therefore, OpenFlow switches can only use a small set of forwarding tables in these

ASIC chips, wasting many other circuits (tables) [1]. Consequently, existing OpenFlow switches only support very few flows in ASIC and resort to tables in SRAM for additional flows. The limited size of ASIC and relatively slow speed of SRAM make OpenFlow switches vulnerable to degraded forwarding speed, which imposes negative impacts on network throughput. Hence, the *data plane performance* of each switch is also crucial to user experience in SDN networks.

Up to date, multiple vendors design and manufacture OpenFlow switches, such as Pica8, Arista, and HP. However, deploying a SDN network is not an easy task because there are many vendor-specific implementations to choose from, and each switch costs a lot. In addition, deploying a new network is a labor-intensive process and requires much time for thorough testing and adjustments. Most of the time, network operators have to resort to the (less ideal) *trial and error* approach when deploying SDN networks. This further amplifies the cost of migrating to SDN networks and in turn prevents SDN networks from being widely deployed. One way to address the issue is to thoroughly evaluate the performance of a target SDN network using *simulator (or emulator)*, which reduces the equipment cost and deployment time.

There are several existing SDN simulators (or emulators), but they mainly focus on the data plane performance. Moreover, most of them do not take diverse vendor-specific implementations into considerations in SDN simulations (or emulations). EstiNet [37] is an OpenFlow simulator and emulator that combines the high scalability and realistic data plane packets emulation strengths from both. The authors focus mostly on the network scalability and the accuracy of data plane performances in large networks. They claim to have better data plane performance accuracy compared to Mininet [36], but EstiNet is a proprietary solution, while Mininet is a large-community open source project and can be easily setup in PCs or laptops. But Mininet authors themselves dictate the lack of performance fidelity due to the limitations of current implementations [26]. However, both Mininet and EstiNet fail to consider control plane performance accuracy and switch diversity.

In this thesis, we aim to develop a detailed and accurate OpenFlow emulator based on the open source projects, Mininet [2] and Open vSwitch (OvS) [5]. We divide the tasks into two steps: (i) deriving switches' performance models, and (ii) integrating the derived models with Mininet and OvS. In the first step, we design several test scenarios based on the switch states or traffic patterns that may affect switch performances and study the impact of them, such as existing flow sizes or data plane packet sizes. The measurements are also conducted on several switches to obtain unique sets of switch-dependent parameters that characterize different switch implementations. The parameters with switch states are taken as performance modeling inputs for accurate performance emulation. In the second step, we need to augment Mininet and OvS into a detailed emulator by integrating our performance models. In the emulator, we need to extract and maintain necessary switch states and read from different switch-dependent parameter configuration files to be capable of emulating various OpenFlow switches, and record the controller-to-switch OpenFlow event information and timestamps to statistics files.

1.1 Contributions and Organizations

This thesis makes the following contributions.

- **Switch performances benchmarks.** We propose automatic measurement procedures for both control and data plane performances. Measurements can be conducted on any OpenFlow switches, and switch performance statistics, such as flow table update delays of different *flow_mod* commands and packet forwarding latency, are obtained and recorded. Switch-dependent parameters are derived from the measurement results and store for further use in our proposed performance models for emulation of different switches.

- **Performance models.** From extensive measurement studies on control and data plane performances, we propose several switch performance models: flow insertion, flow modification, and flow deletion time models of control plane performances, and packet forwarding latency model of data plane performances. They take inputs of switch states or traffic patterns with switch-dependent parameters to emulate diverse implementations of OpenFlow switches.

- **Emulator implementation with performance models integrated.** We integrate our performance models into a popular open source emulator, Mininet/OvS. By extracting *flow_mod* command information from controller-to-switch messages and maintaining switch states in our emulator implementation, we are able to model performances by taking the real-time switch information and switch-dependent parameters as performance model inputs.

The rest of the thesis is organized as follows. We survey the literature in Ch. 2. Ch. 3 presents our measurement methodology and testbed setup. Design of our test scenarios for control and data plane modeling are described in Sec. 4.1 and 5.1, respectively, and each is followed by a detailed study on measurement results in Sec. 4.3 and 5.2. Control and data plane models are presented and explained in Sec. 4.4 and 5.3. Ch. 6 presents the design and implementation of our emulator, and the evaluation results. Conclusions and future works will be discussed in Ch. 7.

Chapter 2

Related Work

With the rise of SDN, several papers [15, 25, 30, 39] gain insight into this new network architecture. They investigate extensive works, provide overviews of recent researches, from infrastructure layer to application layer, and conclude with several promising research directions to work on. To leverage SDN benefits of programmability, flexibility, and efficiency on network management, researchers work on developing and deploying innovative network services, such as multicast, load balance, access control, and network security applications. Experiments for verifications and evaluations of those developed applications are either done with real testbeds or emulations (or simulations).

2.1 OpenFlow Emulators/Simulators

There are several emulators (or simulators) available and easily accessible for researchers to setup for network verifications and performance evaluations. In this thesis, we focus on both control plane and data plane performance accuracy and switch diversity in OpenFlow emulations.

NS-3 [21], originally a well-developed and modularized network simulator, is extended to support OpenFlow standard [3] in order to provide OpenFlow simulations. NS-3 runs OpenFlow simulation as a single process, so it is not difficult for NS-3 to achieve better scalability. But the drawback is that there is no realistic data plane traffic, so the performance model becomes important for accurate performance simulation, but NS-3 uses a relatively simple model. In addition, NS-3 only supports OpenFlow version 0.8.9 and does not enable the use of external controllers. Similar to NS-3, FS-SDN [18] extends from existing work, FS [35], which uses discrete-event simulation implementation and adopts an existing TCP throughput model for accurate measurements. But FS-SDN focuses on data plane performance accuracy and network scalability.

Unlike NS-3 and FS-SDN, Mininet [19], an OpenFlow emulator, emulates each network instance using light-weight containers and sends realistic data plane traffic. Mininet is widely-used among researchers due to its easy accessibility; it can be setup simply on PCs or laptops. However, the authors themselves dictate the performance fidelity issue owing to the limitation of their implementations [26].

On the other hand, Estinet [37], which is an OpenFlow simulator and emulator, combines strengths from both. They modify kernel implementation to directly manipulate on clock time for achieving better performance fidelity. The authors claim to have better performance fidelity and network scalability than Mininet [36]. Data plane performance and network scalability are main focuses of their implementations, but switch diversity is not considered. In addition, EstiNet is a proprietary solution, while Mininet and OvS are both open source projects with large communities, so we decide to develop our detailed OpenFlow emulator based on Mininet/OvS.

2.2 Techniques to Improve Simulation/Emulation Fidelity

As previously mentioned, Mininet [19] authors reveal a performance fidelity issue in their early release. The authors make improvements on performance fidelity and release in version 2.0 [20] via resource isolation of virtual hosts, switches, and links in the emulated networks. In their work, they aim to achieve realism on the functionality and timing, but they mainly focus on the data plane performance fidelity such as Round Trip Time (RTT) and throughput, and evaluate and compare the performance results with real hardware setup. They do not consider diverse switch implementations, either.

In [27], the authors discuss on the diverse switch implementations. They indicate that with different implementations, such as the use of flow tables and flow installation behaviors, both data plane and control plane performances can differ from one another in the same setup. For instance, Ternary Content-Addressable Memory (TCAM), special hardware to store flows used in several hardware switches, is different from a software-implemented flow table in flow installation and flow lookup behaviors, and performances thereby. In this thesis, we aim to develop accurate performance models that capture the various characteristics of vendor-specific switch implementations.

On the other hand, Huang et al. [22] also try to reproduce the proper behavior of different implementations of OpenFlow switches by adding a proxy, which imposes extra delays according to statistical results from different switches, between the controller and OvS. In their work, they consider only the control path delays of data plane performances, i.e., the *PacketIn* path delays between the controller and the switch. In contrast, we manage to improve emulation fidelity by proposing performance models based on extensive measurement analysis on both data plane and control plane.

2.3 Performance Measurements and Modeling of SDN Switches

Several performance measurements are conducted to evaluate and compare the performances of conventional networks and SDN-based networks. By performance comparisons, they examine performance degradation and possibilities of deploying SDN networks in replace of conventional networks. Gelberger et al. [17] and Bianco et al. [13] both measure and evaluate data plane performances in terms of latency and throughput using different workloads, and compare among conventional networks and different SDN architectures, OpenFlow and ProGFE. Emmerich et al. [14] conduct performance measurements on Open vSwitch and provide observations of switch performances in various aspects. Shibuya et al. [34] propose a solution to measure performances of all physical links in SDN-based networks from a single point.

Most of the above measurements are conducted on switch data plane. But we consider the control plane performance, which is the delays of updating flow entries in the flow table, is as important as data plane performance. Rotsos et al. [33] propose a framework for switch performance evaluations and also enable users to develop customized testing modules based on their framework. The authors conduct several sample experiments in their paper, such as delays of flow table update and flow statistics polling. We manage to develop our measurement tools based on this framework.

Kong et al. [24] present OFSim that simulates real-life ISP traffic with multiple switches and one controller and examine whether current implementations of SDN and OpenFlow can meet the requirements of running ISP traffic. They reveal bottleneck may be in flow installation time. Despite measurements, they propose simple models with configurable parameters obtained from analytical results for their simulator though they do not consider the different conditions of switches or traffic patterns that affect on switch performances in their models.

In [23,28], the authors measure data plane performances and model switch forwarding architecture using queuing model. They focus on the packet sojourn time (i.e., how long the packet stays in the switch or controller). They evaluate sojourn time estimation using different forwarding possibilities and different data workloads. Similarly, Azodolmolky et al. [11] also conduct measurements on data plane and model SDN architecture using network-calculus-based analytical model. Both of the models fail to consider different switch implementations, which are going to be addressed in this thesis.

Chapter 3

Measurement Methodology

OpenFlow switches from different vendors have different implementations, and the control plane and data plane performances differ from one another due to the different implementations. Our performance models aim at not only accurately modeling both control and data plane performances, but also deriving a set of parameters that can capture the characteristics of different OpenFlow switch implementations and simulating various switches by adjusting the parameter values. A thorough testing and measurement on control plane and data plane performances from different switches are inevitable. Fig. 3.1 shows the high level design of our measurement work flow.

First of all, we need to conduct measurement studies on several representative OpenFlow switches: hardware OpenFlow switches from different vendors, such as Pica8, Arista, and HP, and software switches, such as OvS and CPqD. In this thesis, we have conducted measurements on two sample switches, *Pica8 P-3297* and *OvS*. Pica8 P-3297 is a hardware switch with ASIC and TCAM design, and it has two-tier tables, TCAM and software-implemented tables, while OvS is a software switch that can be configured and deployed on most PCs, and different hardware specifications of PCs may result in different behaviors. The specifications of two sample switches are given in Table 3.1. In our measurement studies, the relations between switch performances and different switch states, control plane command types, and data plane traffic patterns are considered. Through the measurement studies on control and data plane, our goal is to derive the performance models and switch-dependent parameters according to the observations from the measurement results and knowledge on OpenFlow switch architecture. The measurement studies and

Table 3.1: Specifications of Two Sample Switches.

Switch	**CPU**	**# Cores**	**System Memory**	**Port**
Pica8 P-3297	P2020, 800 MHz	2	2 GB	1 GbE
Open vSwitch	Intel Core i7-2600, 3.4 GHz	4	16 GB	1 GbE

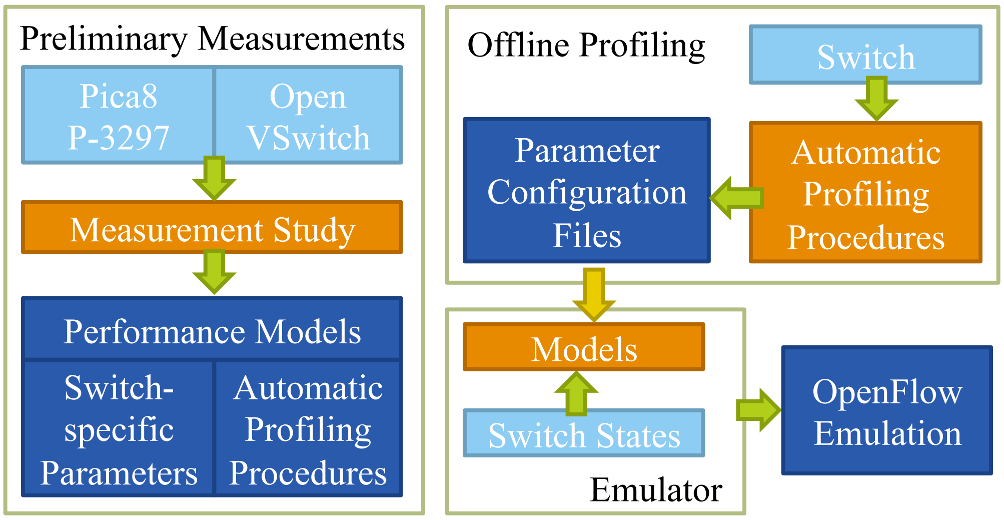


Figure 3.1: Switch profiling and modeling procedures.

results will be presented later in Sec. 4.1 and 5.1 as regards to control and data plane performance measurements, respectively. Sec. 4.4 and 5.3 describe our performance models and how the switch-dependent parameters are obtained from the results. Moreover, those measurements and procedures to derive the switch-dependent parameters will be arranged into automatic profiling procedures.

In the offline profiling step, the automatic profiling procedures are conducted on any OpenFlow switch, and new switch-dependent parameters are derived and recorded in parameter configuration files. In the emulation, the emulator can read from multiple configuration files and use the switch-dependent parameters to simulate various OpenFlow switches, which have already completed the profiling procedures and documented in configuration files. With the parameters and the real time switch states, such as number of existing flows, priority distribution, and traffic patterns as inputs of our performance models, the emulator can emulate data and control plane performances more accurately. Thus, better fidelity simulation can be achieved concerning both the performances and switch diversity.

In the following, we will describe the approaches and setup for our control and data plane performance measurements.

3.1 Control Plane Performance Measurement

For control plane performances, our main concerns are the completion time of *flow_mod* commands. That is, the time required to add, modify, and delete flows in the flow table. Our testing programs should be able to obtain the time when the *flow_mod* command is sent and when is the command is completed so that we can obtain the performances by

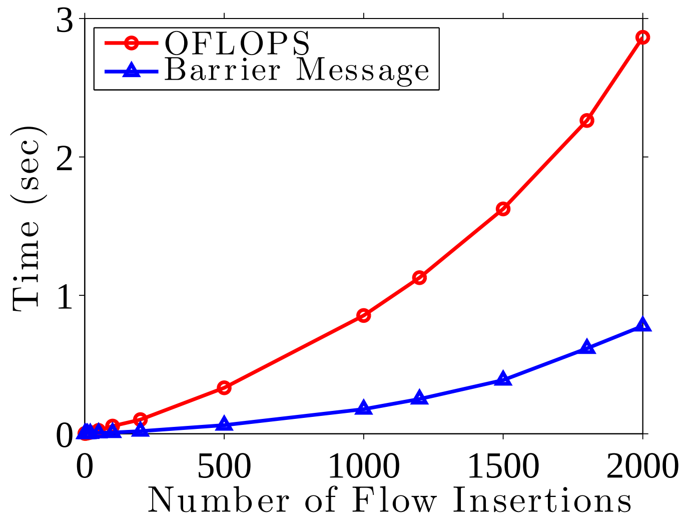


Figure 3.2: Comparison of control plane performance with different measurement methods.

calculating the time differences of *flow_mod* sent time and the *flow_mod* completion time. The *flow_mod* command is sent from our program, so the time can be easily recorded. The problem is how could we ensure that any *flow_mod* command sent to the switch is handled and completed.

One approach is to use *Barrier messages*. In OpenFlow switch specification [6], it defines *Barrier Request/Reply messages* that can be used by the controller to notify any error occurrences as well as the completion of a set of OpenFlow commands. The *Barrier Reply message* should be sent from the switch to the controller after the set of commands sent before the *Barrier Request message* are completed according to the specifications. So we can obtain the time when the command is completed by issuing a *Barrier Request message*. However, the real implementations still depend on switch vendors. But the implementations are usually concealed, so we cannot guarantee it works in all the switches.

A more precise approach may be to directly check if the corresponding data plane packets can be actually forwarded by the flows inserted or modified. Such a platform, called OFLOPS, is presented in [33]. OFLOPS is capable of receiving information from data plane and control plane channels at the same time. OFLOPS works as an OpenFlow controller with OpenFlow protocol 1.0 implemented, and it is used to examine and verify the implementations of OpenFlow switches. In their paper, they did several switch evaluations, and one of them is to compare the flow insertion delays using *barrier message notification* and *reception of dataplane packets notification*. There is a large gap between these two approaches. We also conducted similar experiments on one of our sample switches, Pica8 P-3297, as shown in Fig. 3.2. Different number of flows ranging from $[1, 2000]$ are inserted, and time calculated from two notification approaches are

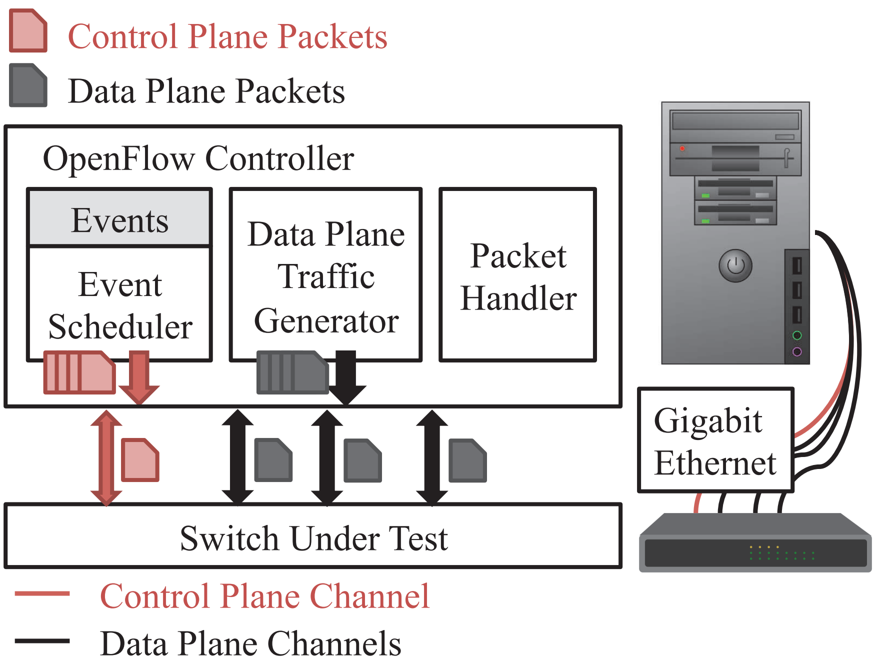


Figure 3.3: Control plane measurement setup.

compared against the other. The result also reveals a large gap between two methods, so the barrier message can not be a good approach as command completion notifications. Since OFLOPS also enables us to develop customized testing modules based on our test cases, so we decide to develop control plane performance measurement tools based on the OFLOPS [4] framework.

Fig. 3.3 shows our control plane performance measurement setup, and there are three main components in the OFLOPS controller. *Event Scheduler* is responsible for handling any event defined in the testing module. We schedule a *Send Flow_mods Event* in our testing module that sends a set of *flow_mod* commands we define to the switch under test. A dedicated *Control Channel* connecting the controller and the management port on switches is responsible for switch/controller message exchange. Once the connection between the controller and switch is established, *Data Plane Traffic Generator* begins to generate and send a set of customized data plane packets that correspond to the flows in the flow table from *sender* via *Data Channels*. Data plane and control plane channels are connected via Gigabit Ethernet. *Packets Handler* will capture and process packets from all network interfaces, both control and data channels. The main responsibilities are to collect packets received from *receivers* to ensure the completion of OpenFlow commands. Upon receiving all corresponding packets, the test terminates, and the time delay is calculated and recorded to files.

3.2 Data Plane Performance Measurement

Considering data plane performances, packet forwarding latency, delay jitter, and throughput should be measured. In data plane performance measurement, we should be able to generate and send customized data plane packets, and record the time when the packet is sent and received. We use pktgen [31] for customized packet generation. We can generate series of packets with specific Ethernet types, source and destination MAC addresses, and source and destination IP addresses. Pcap files are generated and saved in advance based on the test scenarios we design, which will be descibed in Sec. 5.1. Tcpreplay [9] is a Linux utility that can be used to send data plane traffic based on any pre-captured traffic saved in libpcap [8] format. During experiments, we use *OFLOPS* to insert necessary flows and then use *Tcpreplay* to send data plane traffic based on the pcap files pregenerated by *pktgen* for data plane measurement. Sent and received timestamps of data plane packets are recorded through *tshark* [10], a command line utility of Wireshark [38], that can listen and capture packets on any designated network interfaces.

Chapter 4

Control Plane Performance Modeling

We conduct measurement studies for control plane performance modeling on flow insertion time, flow modification time, and flow deletion time.

4.1 Test Scenarios

4.1.1 Factor Considerations

In [27], the authors consider switch diversity important when developing a novel SDN control system. They conduct several experiments and have detailed discussions on diversified characteristics of OpenFlow switches. This motivates our design of the following test scenarios. Speaking of the control plane performance, they mention that different orders of different types of flow commands on different switches diversify the time delays for updating the flow table, so we need to analyze the performances of separate commands. In addition, TCAM in most hardware switches is required to update flow entries in priority order, so we should also consider the impact of different priority distribution of existing table flows along with the existing number of flows.

We design a series of test scenarios based on the following factors.

Priority distribution. As for table like TCAM, priority plays an important role in updating flow entries in the flow table. Therefore, we design three different priority distributions, ascending, descending, and same priority distribution, for measurement studies.

For adding new flows into the table, the priority distribution mean the priority order we used. For example, ascending priority and descending priority means that we send the *flow_mod* commands with the priority value increasing or decreasing by one every subsequent command. Ascending priority order starts from the priority value 65535 while descending priority order starts from 501. Same priority order uses only one priority value 501 for all flows inserted. Fig. 4.1(a) shows an example of adding three new flows using

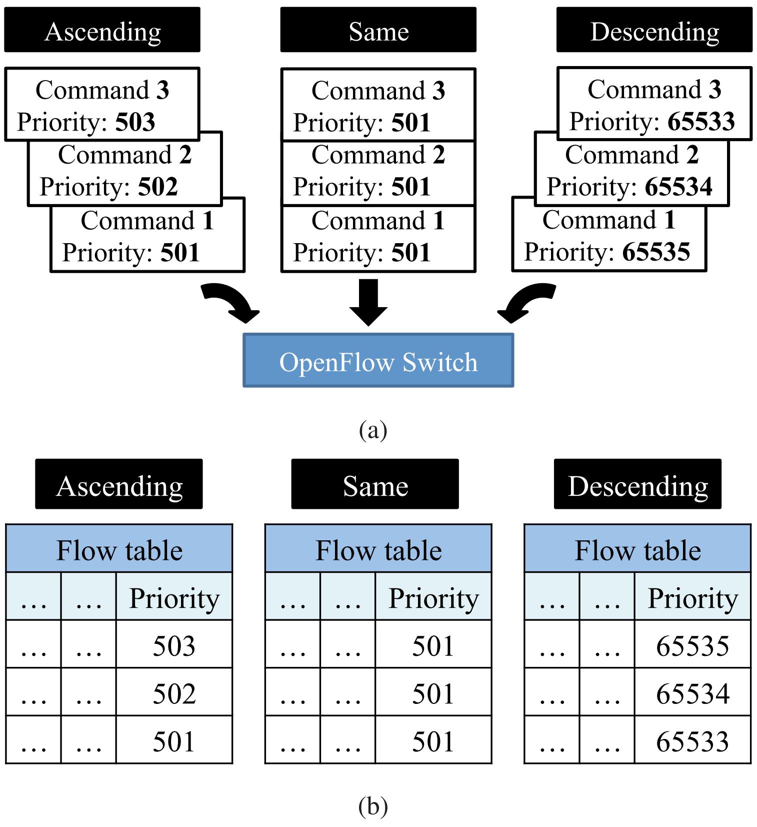


Figure 4.1: Example of different priority distributions when: (a) adding 3 flows and (b) 3 flows in the flow table.

these three priority distributions. For modifying or deleting existing flows in the table, different priority distribution means the range of priority values of flows in the table. Fig. 4.1(b) shows an example of three flows in the flow table in three different priority distributions.

Flow with higher priorities should be matched first, so the flows must be kept in the priority order in TCAM. For descending and same priority distributions, no higher priority flows are inserted each time, but for ascending priority distribution, the new flows are come with higher priorities each time, so the existing flows in the table should be shifted for higher priority flows to be put into the table. Therefore, flows with higher priority values are kept in the top position, as shown in Fig. 4.1(b).

Notice that we avoid using priority values no greater than 500 since we consider that priority values within the range $[0, 500]$ may be kept for special uses in some OpenFlow switch implementations.

Number of existing flows. For switches such as OvS, software tables are used. Different number of flows in the table may affect the time for searching and updating a flow in the flow table. Yet, we are also not sure about its impact on other implementations of flow tables, so this should be taken into considerations. We will test the time delay of three commands under different number of existing flows in the flow table, which ranges from $[1-7000]$.

Number of batch commands. Switch implementations may manage to gain benefits from batching a number of commands and reduce time from optimizing processing procedures of those commands, such as command order rearrange. We measure the time delays with different numbers of OpenFlow commands sent back to back per time. The intervals between commands are short enough under the time-out time for separating commands in different batches. The command size ranges from $[1-100]$. We also manage to find out the maximal batching size from those measurement studies.

4.1.2 Scenarios

```
1: for each priority distribution $\omega = desc, asc, same$ do
2:     for each existing flow size $e = 1, 5, 10, 20, 50, 100, 500, \ldots, 7000$ do
3:         for each batch command size $q = 1, 3, 5, 10, 20, 50, 80, 100$ do
4:             insert $e$ flows with priority distribution $\omega$, output port 2, and record the
                   time delay
5:             randomly select $q$ flows, modify the output port to port 3, and record the
                   time delay [Flow modification scenario]
6:             insert another $q$ flows with priority distribution $\omega$ and output port 2, and
                   record the time delay [Flow insertion scenario]
7:             delete $q$ flows so that there are $e$ flows in the table
8:             send deletion command that match all flows in the table and record the
                   time delay when the barrier_replay message is received [Flow deletion
                   scenario]
9:         end for
10:    end for
11: end for
```

Figure 4.2: The pseudo code of the control plane measurement procedures.

The flows we use in the measurements are configured with following fields: *ingress port*, *Ethernet type*, *destination MAC address*, *destination IP address*, and *egress port*. Following are the three test scenarios we design in our measurement study.

Flow insertion scenario. We first preinstall a number of flows to setup the switch flow table with different existing flow size. Then, we send flow insertion commands under different existing flow size and measure the time to complete the commands. Average time is calculated as per flow insertion time. Different priority orders are tested to study on the impact of flow shifting and differences between different priorities and same priority test performances. Different numbers of insertion commands are also tested to study on any batching effects.

Flow modification scenario. Similar setup to the flow insertion time test scenario, we send flow modification commands under different existing flow size and measure the time for command completion. Average time is calculated as per flow modification time. Different priorities and same priority cases for flows in the flow table are tested. Different number of flow modification commands are also tested as we have designed for insertion tests.

Flow deletion scenario. For deletion time, we also preinstall a number of flows. Then, we send a deletion command that matches all flows in the flow table. The total time cost is measured with different number of flows deleted. The notification of operation completion is through *barrier messages*.

In our measurement results, each sample is derived from the mean value of 10 runs of measurements with the same setup, and the 95% confidence intervals are calculated as errorbars shown in our measurement results.

Control plane performance profiling procedure. Fig. 4.2 shows the automatic profiling procedure of control plane performance measurement. We first add e number of flows into the flow table and record the time delay to add e flows. Then, q flows are selected and modified, and the time delay for modifying q flows under existing flow size e is obtained. We add another q flows and record the time delay for adding q flows under existing flow size e. By this time, there are $(e + q)$ flows in the flow table. In order to measure the time delay for deleting e flows, we randomly select q flows and delete them from the flow table. At last, a wildcarded deletion command that matches all flows in the table is sent and the time delay for deleting e flows is recorded.

4.2 Additional Measurements

4.2.1 Multiple Flow Tables

Aside from test scenarios in 4.1, we have also conducted several experiments for switches with multi-level flow tables. Fig. 4.3 shows time delays for adding different number of flows to Pica8 P-3297, and the result reveals that the growing trend changes in between existing flow size of 4000 to 4500, which indicates a fact that different tables are used so as to result in different behaviors and thus different insertion performances.

We manage to include usage of multiple tables into our performance models. We consider the multi-level table structure as similar to the cache architecture. Therefore, there are two main issues that need to be addressed. One is the flow table size since we

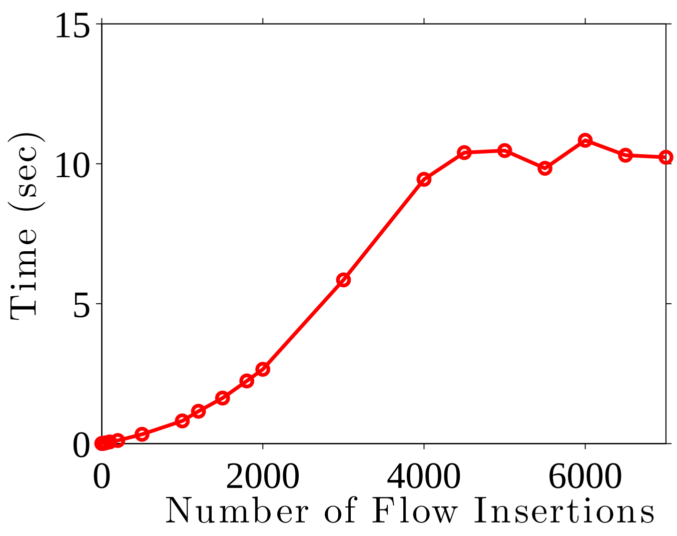


Figure 4.3: A sample result of flow insertion time with different number of flows inserted.

need to know when the first level of the table is full and when the higher-level tables are used. The other is how flows are populated in multiple tables; we refer it as the cache algorithm. It is necessary since we need to know which tables are involved for every OpenFlow commands so that we can accurately model the performance.

Flow table size inference. As the Fig. 4.3 indicates different trends happen before and after existing flow size of 4000 and 4500. The first level flow table size should basically fall into the range $[4000, 4500]$, and there are two levels of flow tables. A possible approach to obtain a more exact number of table limit is to send a bunch of flows over 4500 to fill up the cache. Then, we send one or two data plane packets per flow and record the packet forwarding latency. After the results are collected, we can do clustering based on the forwarding latency using clustering tool like *K-Means*. The cache table size can be derived from the cluster size with smaller values of the cluster center (i.e., shorter latency).

Cache replacement algorithm inference. Another issue is that cache algorithm should be inferred and included in performance models with multi-level tables. We consider following flow attributes as possible cache algorithms that decide which flows should be discarded when the cache (i.e., first-level flow table) is full.

- **Last modification timestamp.** The timestamp should indicate when the flow is inserted into the flow table or when it is last modified if any.
- **Last used timestamp.** The timestamp records the time when the last data plane traffic matches the flow for packet forwarding.
- **Traffic counts.** All matched data plane packets of the flow should be counted in bytes and record as traffic counts.

```
1:  let policies $W = used, modified, traffic, priority$
2:  let unique policies $X = modified$
3:  let $T$ be the cache size
4:  let $CA$ be an empty array
5:  while $CA$ do not contain members in $X$ do
6:      add $2 \times T$ flows with random priorities in $1^{st}, 2^{nd}, 3^{rd}, 4^{th}, 5^{th}, 6^{th}, 7^{th}, 8^{th}$ partition orders
7:      send data plane traffic in $1^{st}, 2^{nd}, 5^{th}, 6^{th}, 3^{rd}, 4^{th}, 7^{th}, 8^{th}$ partition orders with packet size $400, 200, 400, 200, 400, 200, 400, 200$ bytes
8:      send probing traffic and record the packet forwarding latency
9:      cluster into multiple partitions based on the latency
10:     find the relation between the cluster and attribute coverage, and determine the cache algorithm with the highest relation
11:     add the cache algorithm to $CA$ array
12: end while
```

Figure 4.4: The pseudo code of an example of cache algorithm inference experiments.

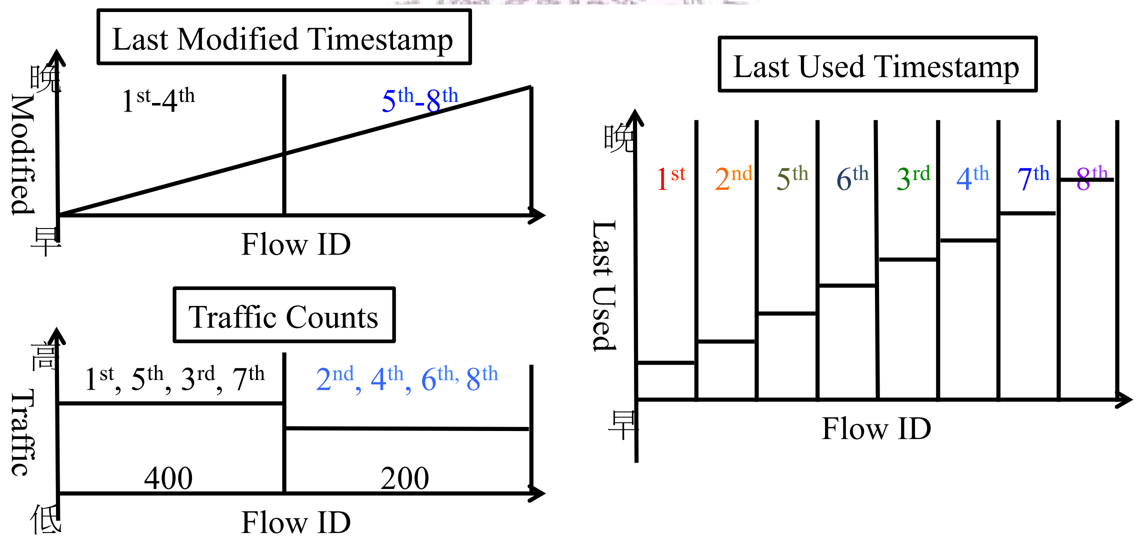


Figure 4.5: Different coverage of replacement choices using different cache algorithm.

- **Priority.** Priority value of the flow.

We consider both increasing and decreasing orders for those attributes. For example, for last modification timestamp, we examine the possibility of cache algorithm for top newest timestamps (decreasing order) or oldest (increasing) timestamps. We also infer the order of those attributes when attribute values are the same. That is, if we infer the cache algorithm as traffic counts for instance, when the traffic counts are the same, we need another attribute as cache algorithm. This is referred to as the order of the attributes.

There are three main steps for our cache inference experiments. Fig. 4.4 shows an example of the inference experiment. First, we insert twice the cache size of flows with random priorities. We separate the inserted flows into 8 partitions for following steps. Second, we send data plane packets for one partition per time to make flows in different partitions with different last used timestamp and traffic counts, so different attributes should result in different coverage of partitions. For instance, flows are inserted following the order from 1^{st} to 8^{th} partition, so the last modification timestamp in increasing order is $\{1, 2, 3, 4, ..., 8\}$. We send data plane traffic in partition orders of $\{1, 2, 5, 6, 3, 4, 7, 8\}$ with traffic counts of $\{400, 200, 400, 200, 400, 200, 400, 200\}$ bytes, respectively. The last used timestamp in increasing order is $\{1, 2, 5, 6, 3, 4, 7, 8\}$, and the traffic counts in increasing order is $\{2, 4, 6, 8, 1, 3, 5, 7\}$. All attributes in both orders cover different combinations of partitions as Fig. 4.5 shows. The last step is to send probing data plane traffic under such condition and record the packet forwarding latency. We cluster the latency to separate them into different tables and calculate the relations of the cluster with higher latency (i.e., the flows to be discarded from the cache) and the corresponding partition coverage of each attribute in increasing or decreasing orders. The attribute with the highest relations will be considered the cache algorithm.

To determine the next order of the attribute, we conduct the same experiment again, but we keep the values of previous order of the attribute the same to exclude the attribute from effects. For instance, if priority with increasing order is decided as the first cache algorithm, the next experiment will insert flows with same priority in order to examine only the impact of the rest attributes. Note that the experiment will end when the attribute, last modified timestamp, is selected since each flow has a unique last modified timestamp value.

Obtaining both the cache size and cache algorithm from above experiments, we can conduct the three aforementioned test scenarios, insertion, modification and deletion time tests for the existing flow size over cache size. By using the cache algorithm, we can control the insertion, modification, and deletion commands to occur in the second or later level of flow tables. Then, the later level flow table performances can be successfully measured in our measurement studies and considered into control plane performance models.

4.2.2 Batch Commands

We assume that there may be optimization implementations for command batching, and we also observe it in the measurement result described later in Sec. 4.3. There should be two issues for command batching. First question is that when should the command batching happen. Commands arrive at different time, and we consider there is a countdown timer for deciding whether to collect commands for batch processing. The commands are collected as same batching before the timer times up, and the timer should reset when new commands arrive before time up. All commands collected should considered as same batch and will be processed together for performance optimization. However, if there are hundreds of commands come in short time and collected as same batch, switch will be very busy processing those commands, which may starve other processes on the switch, such as switch statistics checking or reply of hello messages from the controller. This may cause problem to the switch; for instance, if the switch cannot reply a hello message back in time, the controller may consider the switch to be disconnected or something. So there should be a maximal size for batch processing.

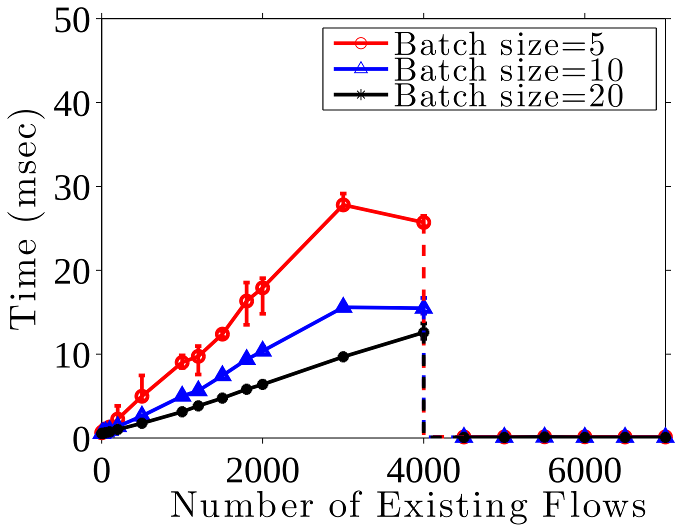
Time-out time. We conduct experiments to send control plane commands with different inter-packet time by inserting sleep between two consecutive commands. We start from 0 inter-packet time and increase by 100 microseconds each time until we observe a sudden jump of the commands completion time. The threshold is recorded as the batching timer.

Maximal batch command size. For maximum command batching size, in test scenarios described in 4.1, we conduct experiments to send different number of insertion commands, ranging from 1 to the flow table size. We calculate the per insertion command processing time by taking the average values. We examine the per insertion command processing time and find the minimum time. We take the number of insertion commands as the maximal batching size at the minimal time value.

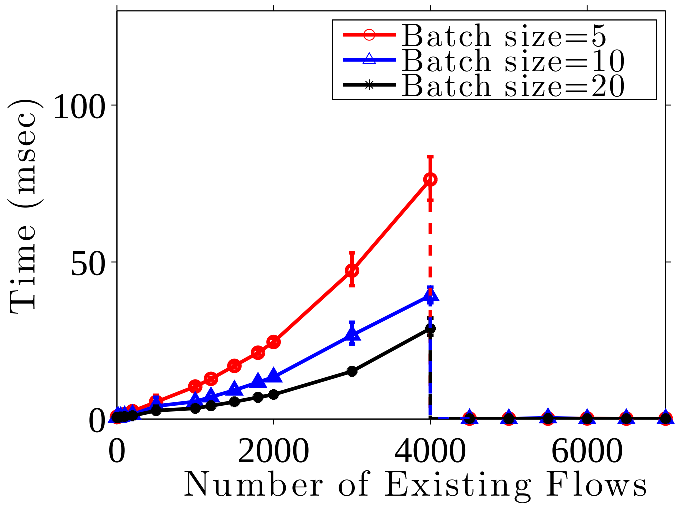
4.3 Measurement Results

4.3.1 Results of Sample Switch, Pica8 P-3297

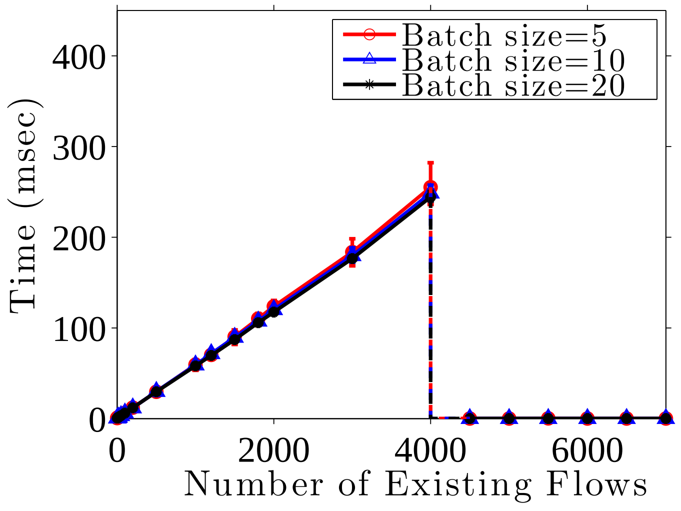
Per flow insertion time grows linearly with the increase of existing flow size, but increasing rate depends on the priority distributions and batch command size in TCAM. We observe that the per flow insertion time grows linearly along with the increase of the existing flow size for different priority distributions, as Fig. 4.6 show. But among them, the insertion time of ascending priority distribution grows much faster than the other two since it requires more time to rearrange existing flows and new inserted flow in priority order. The performances of same priority and descending priority distribution, on the other hand, grow in a much slower rate. We consider the growing trend as the processing overheads from switch implementations, and performances for single priority and multiple priorities are different, as well. For same and descending priority order, with more flow insertion commands, the per flow insertion time grows less, and we consider it as optimization benefits from batch processing. For ascending priority, since the shifting



(a) Same priority distribution.

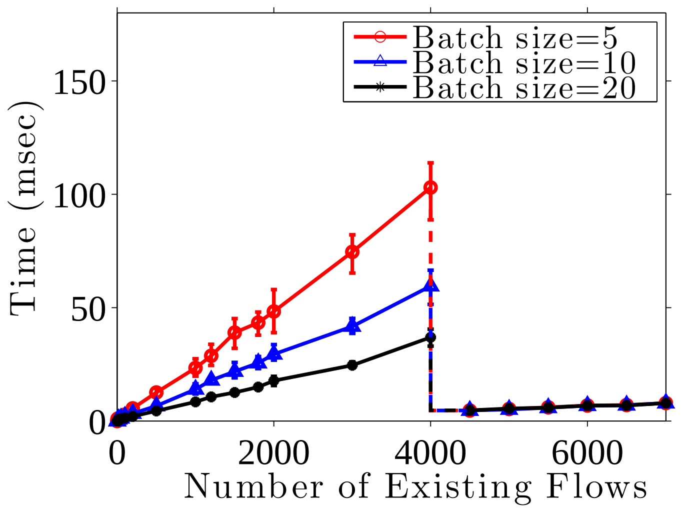


(b) Descending priority distribution.

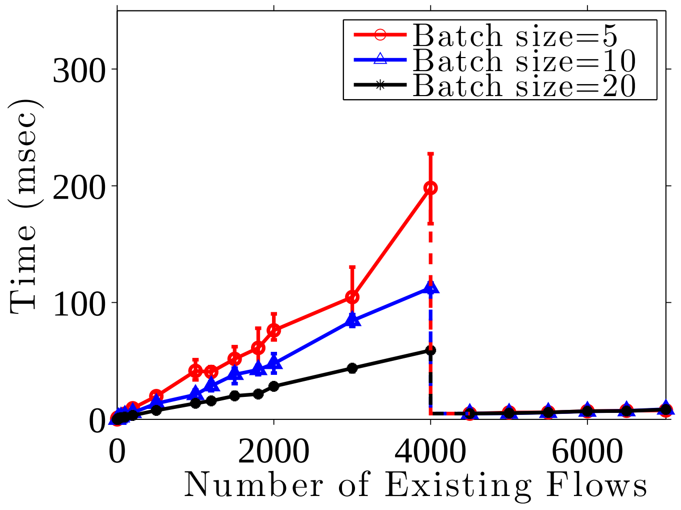


(c) Ascending priority distribution.

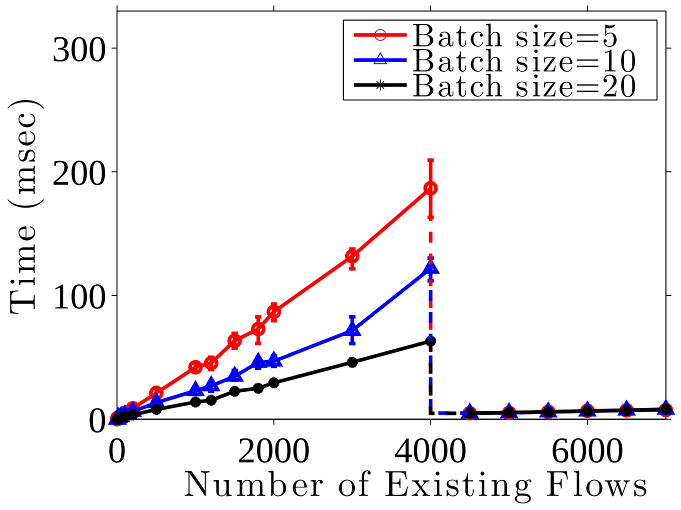
Figure 4.6: Sample results for flow insertion time on Pica8 P-3297.



(a) Same priority distribution.



(b) Descending priority distribution.



(c) Ascending priority distribution.

Figure 4.7: Sample results for flow modification time on Pica8 P-3297.

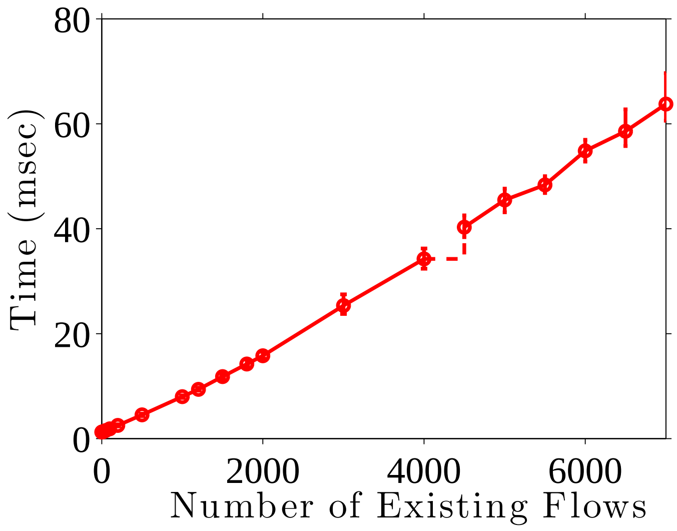


Figure 4.8: A sample result for flow deletion time in ascending priority distribution on Pica8 P-3297.

time dominates the time delays, there are not much differences among different batch command size.

Per flow insertion time is constant, but the values depend on the batch command size in the software table. For results in the second level table (i.e., the software table of Pica8 switch), the performances are independent from the existing flow size. But batching effects still can be observed from the results; it takes less time for per flow insertion when more commands are sent back to back per time. In addition, different priority distributions have little impact on the performances, which means the flow shifting is not needed in the software table.

Per flow modification time grows linearly with the increase of existing flow size in TCAM and software table. The modification time as in Fig. 4.7 show, the time grows along with the increase of existing flow size for three different priority distributions. We refer the increasing rate as processing overheads of flow modification. Different increasing rates are observed between single priority and multiple priority distributions (i.e., ascending and descending priority distributions). In our model, we will separate and generate model parameters under single and multiple priority conditions. As similar to what we observe in per flow insertion time results, per flow modification time also takes benefits from batch processing. For results in the software table, the behavior is similar to what we observe in TCAM although the time delays are much less.

Flow deletion time is proportional to the number of flows deleted in TCAM and software table. Since different priority distributions do not matter much for flow deletion time, here we show the sample result of existing flows with ascending priority distribution in Fig. 4.8. With more existing flows, it requires more time to delete all flows from the flow table no matter it is in TCAM or software table.

4.3.2 Results of Sample Switch, Open vSwitch

Since the flow insertion with different priorities are installed using hashing in OvS, different priority distributions basically do not matter the flow insertion time. The results are as we expect, and for the results of modification and deletion test scenarios, priority distributions also have little impact. So here we take the descending priority distribution as sample results for three test scenarios as shown in Fig. 4.9.

Flow insertion time is constant, but the constant time depends on the batch command size. Since the flow table implementation is done with hash table, and there is no flow shifting required to put new flows into the table with greater priority as it does in TCAM of Pica8 switch, the per flow insertion time is constant under any number of existing flows as Fig. 4.9(a) shows. But with higher batch command size, the constant time decreases.

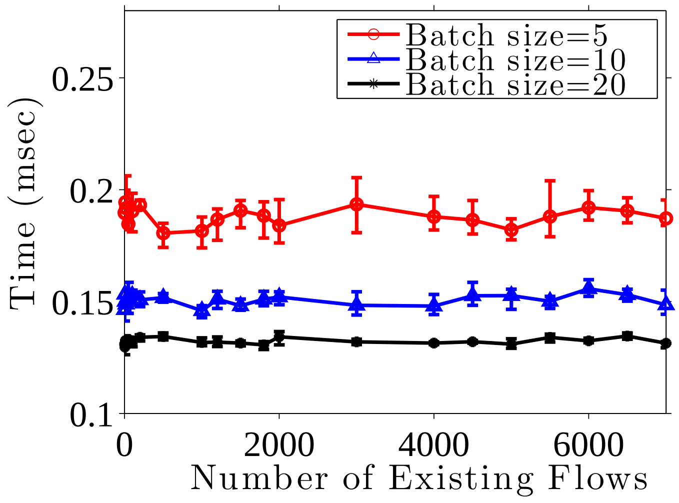
Flow modification time grows linearly with the increase of existing flow size. As similar to the Pica8 results, we consider the increasing as the processing overheads. The increasing rates also depend on different batch command size. With more batch command size, the increasing rate decreases as Fig. 4.9(b) shows.

Flow deletion time grows linearly with more number of flows deleted. More number of flows require more processing time to delete flows so the time is proportional to the number of flows deleted as Fig. 4.9(c) shows.

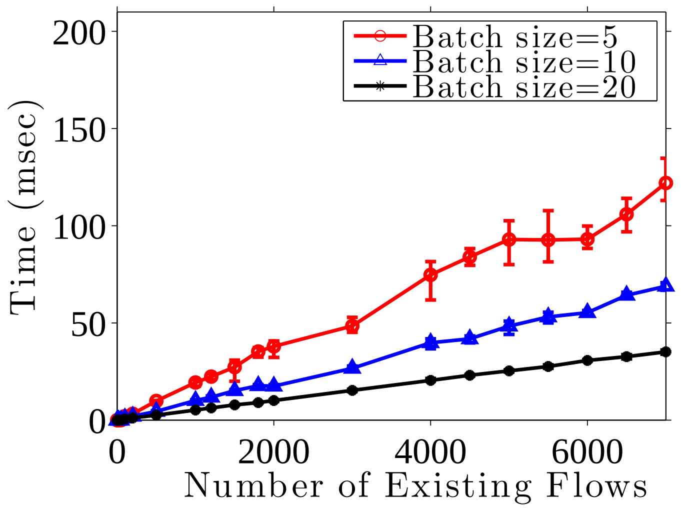
4.4 Performance Models

OpenFlow specification defines three main flow table modification commands, insertion, modification, and deletion of flow entries. Different commands on different switches may require different processing procedures. For inserting flows in flow tables like TCAM, it takes time to arrange flows in priority order. We refer the arrangement overheads as the *shifting time* since it requires to shift existing flow entries to make vacant spaces for new flows entries with higher priorities. Moreover, in regards of modification and deletion of existing flows, there is overhead to search table entries and collect all matched flows. We refer the overhead to be the *matching time*. And the basic flow table modification overhead (without all preprocessing overheads like flow shifting or flow matching, only the action to write changes to the table) is referred to as *operation time*.

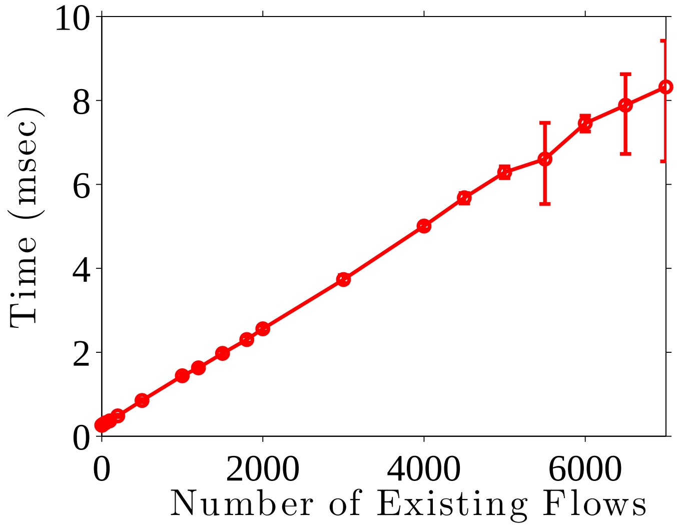
From analysis of measurement results of Sec. 4.3, we consider following as our modeling inputs. Table 4.1 shows the symbols we use in this thesis.



(a) Flow insertion time.



(b) Flow modification time.



(c) Flow deletion time.

Figure 4.9: Sample results on Open vSwitch with same priority.

- **Command c from the controller.** *flow table modification command types,* f_c and the *priority value,* p_c need considering since f_c determines which performance models we should choose, and p_c helps infer the number of shifting times, s.

- **Priority distribution of existing flow entries.** ω We keep track of numbers of flows e_p^t for every priority values p in table t since we need to infer the number of shifting times from the priority distribution and priority value at command c.

- **Number of existing flow entries, e.** As observed in the measurement results, both insertion and modification time is dependent on the number of existing flow entries in the flow table, so we take this into consideration.

- **Number of shifting times, s.** Number of shifting times affect insertion time when different priorities are present. As we observe in the insertion time results of ascending priority distribution, the time grows dramatically because shifts of flows are present in this scenario. We count the number of shifts from the priority distribution ω_c^t and the priority of the coming command p_c. Number of flows with priority that is lower than p_c in the priority distribution ω_c^t should be counted.

- **Number of matching flows, m.** For flow modifications and deletions, a flow table modification commands may match multiple flows in the table, which require more than one write operations, so m is used to infer the number of write operations we need to perform.

- **Number of batch commands, q.** We observe that flow insertion time and flow modification time decrease if we insert (or modify) many number of flows at one time, and we consider it to be optimization benefits from batch processing of flow_mod commands arriving within batch timeout α.

- **Number of flow tables, T.** Number of tables used in the switch. For instance, Pica8 P-3297 uses two tables to store flows, TCAM and software table. We consider different tables as different switch implementations, so we will generate parameters separately for different flow tables from the measurement results of those tables. The parameters can then be used to characterize TCAM and software table, respectively, in our models.

- **Switch-dependent parameters.** Flow shifting S or matching M overheads may differ among different switch implementations. For example, switches may or may not require flow shifting, so the shifting time may be zero or non-zero values. For different flow tables used, matching time also differs. Therefore, we need to define a set of switch-dependent parameters to represent different switch implementations.

We consider different tables used as similar to use another switch, so we derive parameters for different tables, respectively. The parameters for insertion, modification, and deletion models will be detailed described later.

Following we will detailed describe our three models separately.

$$D_{add} = \frac{P^t_{f_c,\omega^t_c}}{R^t_{f_c,\omega^t_c}(q_c)} \times e^t_c + S_t \times s^t_c + W_t \tag{4.1}$$

$$s^t_c = \sum_{i=0}^{p_c} e^t_{c,i} \tag{4.2}$$

Flow insertion time. Eq. (4.1) shows the model for inserting 1 flow into the flow table. P, R, S, and W are the switch-dependent parameters, and t, ω, q, e, and s are model inputs. If multiple tables are present, which flow table to insert the flow should determined first. Table t is determined by the switch existing flow size e_c and table sizes N; if the table is full, higher-level will be used, and the cache algorithm should determine which flow will be moved to higher-level tables.

Observations from Sec. 4.3, flow insertion time is linearly proportional to the existing flow size in both Pica8 and OvS switches, but the increasing rate depends on the switches, flow command types f, flow tables t, batch command size q, and existing flow priority distributions ω. The increasing rate is determined by the base processing overheads, $P^t_{f_c,\omega^t_c}$, and the batch processing decreasing rate, $R^t_{f_c,\omega^t_c}$. $R^t_{f_c,\omega^t_c}$ is a quadratic function with input q_c.

Obviously, f_c is insertion for the flow insertion time model. For priority distributions ω, our model supports two conditions, single priority or multiple priorities conditions, in choosing the P and R values. If commands $i$to $(i + n)$ arrive before the batch command timeout α times up, commands $i, i+1, \ldots i+n$ are included in the same batch with same batch command size equal to $(n - 1)$. $(n - 1)$ should be no greater than the maximal batch command size Q. e^t_c is the number of existing flows in table t.

For result like ascending priority distribution, the time increases drastically since the shifting times increases with the existing flow size (in face, equal to the existing flow size). The shifting times is proportional to the time delays. S_t denotes the time to move one flow entry to the adjacent vacant space, and s^t_c is the number of shifting times required if command c is processed. s^t_c is calculated from priority distribution ω^t_c and the priority value p_c given in Eq. (4.2). One insertion command requires 1 write operation time, W_t.

$$D_{mod} = \sum_{t=1}^{T} \left(\frac{P^t_{f_c,\omega^t_c}}{R^t_{f_c,\omega^t_c}(q_c)} \times e^t_c + M_t + W_t \times m^t_c\right) \tag{4.3}$$

Flow modification time. The modification time model is given in Eq. (4.3). P, R, M, and W are switch dependent parameters, while ω, q, e, and m are model inputs. Per flow modification time also contains processing overheads proportional to the existing flow size in table t, which is determined by P_{f_c,ω_c^t} and R_{f_c,ω_c^t}. f_c should be modification type for sure. As described in the insertion model, ω is used to determined whether it is single-priority or multiple-priority conditions in the current switch state. q_c is also determined as aforementioned.

Time for searching all matched flows in the table M_t should be counted. Multiple flows may be matched if wildcarded values are used. m_c^t denotes the number of matched flows. Write operations should be performed m_c^t times.

$$D_{del} = \sum_{t=1}^{T} (M_t + W_t \times m_c^t) \tag{4.4}$$

Flow deletion time. Eq. (4.4) shows the deletion time model, where M and W are switch-dependent parameters, and m is the model input. Similar to flow modification command, deletion requires time M_t to search all flow entries in table t for m_c^t matched flows in table t. m_c^t times of write operation W_t is needed to remove matched flows from the flow table.

Switch-dependent parameters. We derive the switch-specific parameter values following the steps below. Parameters are generated for different switches and different flow tables. Deletion requires searching for matched flows M and remove flows with m write operations to the flow table. As we observe from the deletion result, the time is linearly increasing with the number of matching flows, i.e., the number of flows deleted. We perform curve fitting using a first-degree polynomial, and obtain function $(a + bx)$. a is the matching time M and b is the write operation time W.

We perform curve fitting on insertion test scenarios with a first-degree polynomial for different batch command sizes. The fitted increasing rate is the processing overheads for different batch command sizes. The base processing overhead, $P_{add,single}$ and $P_{add,multi}$ (where add refers to insertion command type, $single$ and $multi$ are referred to as single priority and multiple priorities), is obtained from the results of batch command size 1 with same priority distribution and descending priority distribution, respectively. The increasing rate decreases quadratically from batch command size 1 to maximal batch command size. The ratios of increasing rate over P for different batch command sizes in $[1, Q]$ are calculated, and curve fitting is performed with a second-degree polynomial. The coefficients of fitted functions are $R_{add,single}$ and $R_{add,multi}$ when same priority distribution and descending priority distribution results are used, respectively.

We expect the differences between descending priority distribution and ascending priority distribution results are on the number of shifting times. Descending priority distribution should be 0 while ascending priority should be equal to the number of existing flows e_t. Therefore, we calculate the mean differences between ascending results and descending results under different number of existing flows and perform curve fitting with a first-degree polynomial. The coefficient of the first-degree term is considered as the S, and S should be non-negative values. If fitted result is negative, then 0 is recorded as S instead.

For $P_{mod,single}$, $P_{mod,multi}$, $R_{mod,single}$, and $R_{mod,multi}$ (where mod of f_c is modification) , similar deriving procedures to the P and R for insertion are used. P is derived from the result of batch command size of 1, and R is derived from the increasing rates of different batch command sizes.

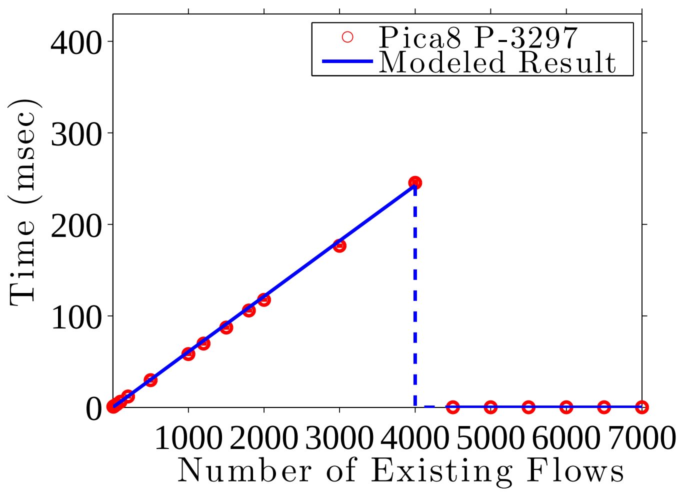
4.5 Model Validation

We have four test scenarios for model validation, insertion, modification, deletion, and random command tests. Figs. 4.10, 4.11, 4.12, and 4.13 show the validation results for insertion, modification, and deletion commands of Pica8 P-3297 and Open vSwitch. The figures show the sample results of using batch command size of 20 with three different priority distributions on Pica8, and the sample results of batch command size of 20 with ascending priority distribution on OvS.

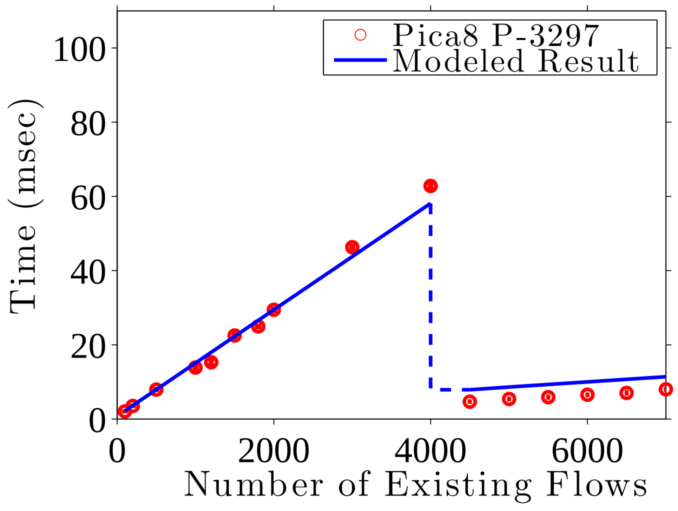
Aside from validations of three commands separately, we have conducted experiments to send random commands (mix of insertions, modifications, and deletions) with random priorities, random arrival time, and random IP addresses. Randomized priorities range from 501 to 65535. We simulate the arrival time using Poisson process with rate 100 since as [12] measurement results suggested, the observed number of flows per second of a switch in the measured networks is at most 10000 flows. Measured networks include campus networks, private enterprise networks, and cloud data centers running different services, and over 10 data centers are measured. We test different number of commands from 100 to 2000 and generate 16 random configurations for each command size. Error rate is calculated by $\frac{abs(modeled-groundtruth)}{groundtruth}$. Mean and 95% confidence interval are calculated from the results of 16 random configurations, and the error rates for different command sizes are shown in Fig. 4.14 for Pica8 and OvS. Expect for command size of 2000 on Pica8 P-3297, the error rates are all under 30%, which is quite acceptable.

Table 4.1: Symbol Table

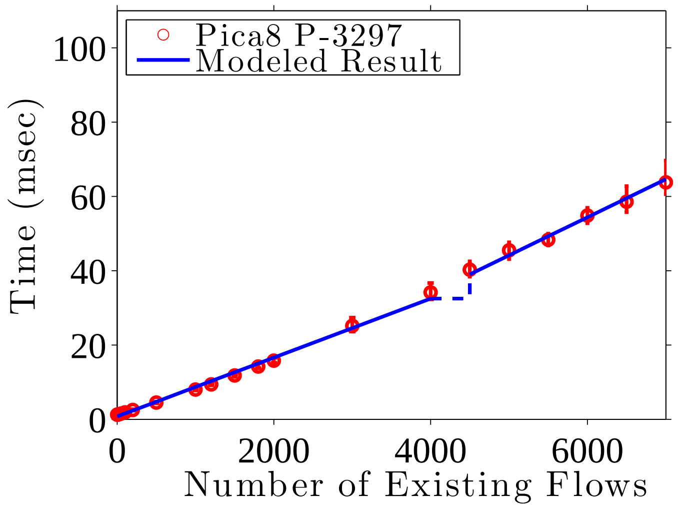
Symbol	**Description**
T	Number of flow tables
t	Index of flow tables
N_t	Flow table size of table t
α	Batch command timeout
Q	Maximal batch command size
c	Index of controller-to-switch flow_mod commands
f_c	Flow_mod command types at command c, including insertion, modification, or deletion
p_c	Priority value at command c
e_c	Number of existing flows in the switch at command c
q_c	Number of batch commands at command c
e_c^t	Number of existing flows in table t at command c
s_c^t	Number of flow shifts in table t at command c
m_c^t	Number of matching flows in table t at command c
ω_c^t	Existing flow priority distribution at command c in table t
$e_{c,p}^t$	Number of existing flows with priority value p at command c in table t
$P_{f,\omega}^t$	Processing overheads for command type f and priority distribution ω in table t
$R_{f,\omega}^t$	Batch processing decreasing rate of processing overheads for command type f and priority distribution ω in table t
S_t	Time for shifting a flow entry to the adjacent vacant space in table t
M_t	Time for searching all matching flows in table t
W_t	Time for writing changes to a flow entry in table t
k	Index of data plane packets
h_k	Matching fields at packet k, including L2, L3, and L2L3
a_k	Inter-packet time between packet $k-1$ and packet k
e_k	Existing flow size at packet k
b_k	Packet size of packet k
A	Base inter-packet time
E	Base existing flow size
B	Base packet size
Δa	Level differences between inter-packet time a and base inter-packet time A
Δe	Differences between exist flow size e and base existing flow size E
Δb	Differences between packet size b and base packet size B
β_h^t	Base forwarding delay time in table t at base inter-packet time A, base existing flow size E, base packet size B, and matching field h
γ_a^t	Increasing rate of forwarding delay in table t for different inter-packet time
γ_e^t	Increasing rate of forwarding delay in table t for different exist flow sizes
γ_b^t	Increasing rate of forwarding delay in table t for different packet sizes



(a) Per Flow Insertion Time Comparison.

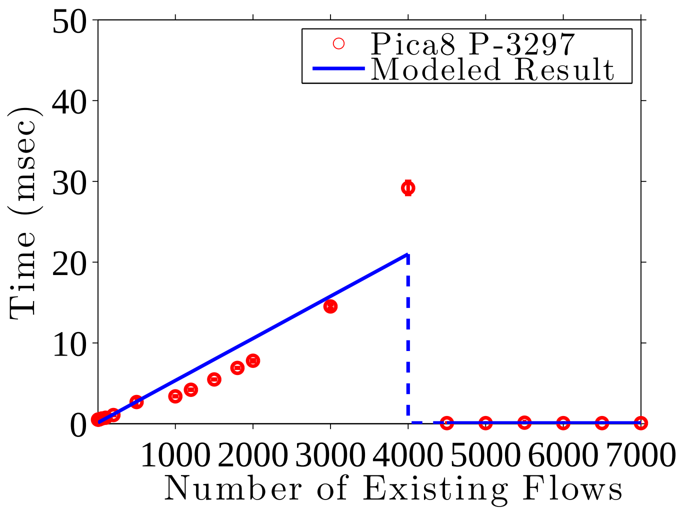


(b) Per Flow Modification Time Comparison.

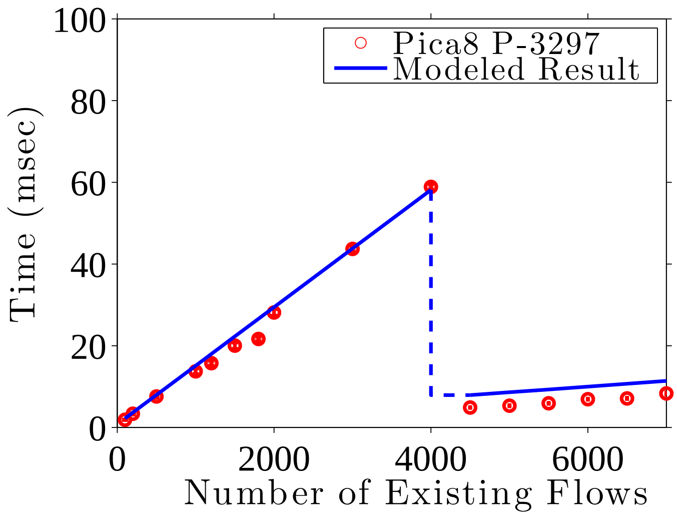


(c) Deletion Time Comparison.

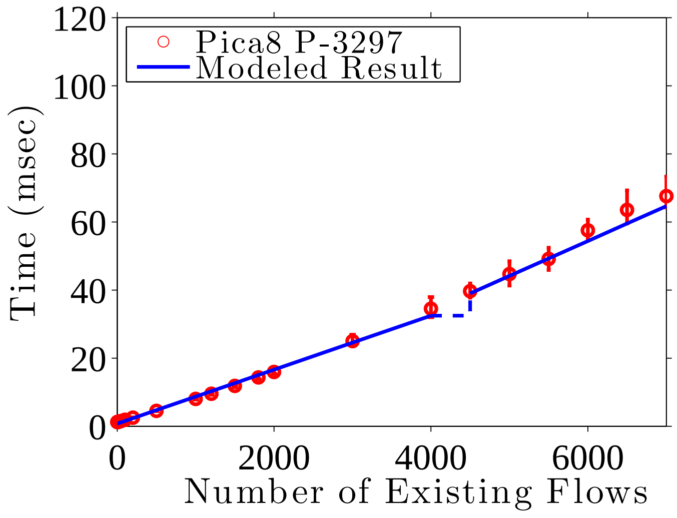
Figure 4.10: Validation results on insertion, modification, and deletion time with ascending priority distribution.



(a) Per Flow Insertion Time Comparison.

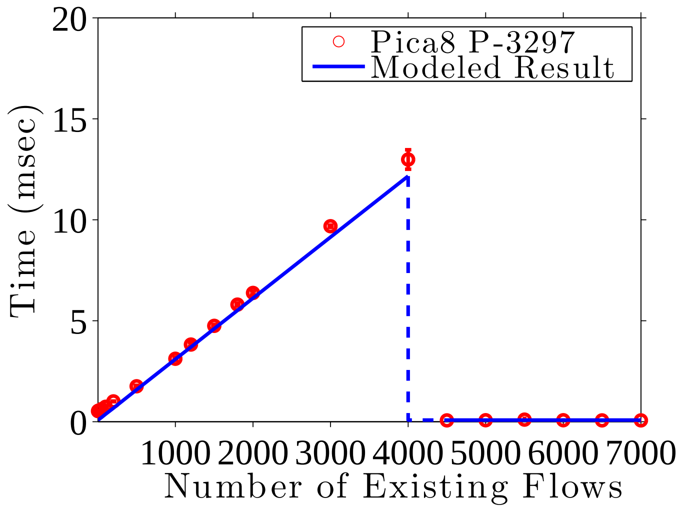


(b) Per Flow Modification Time Comparison.

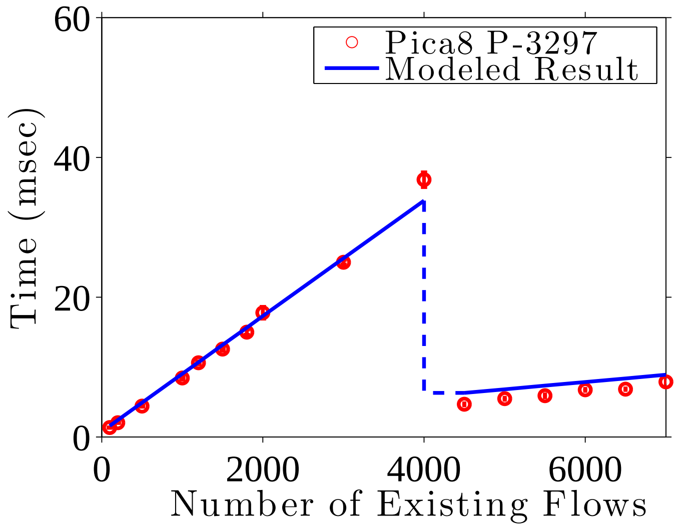


(c) Deletion Time Comparison.

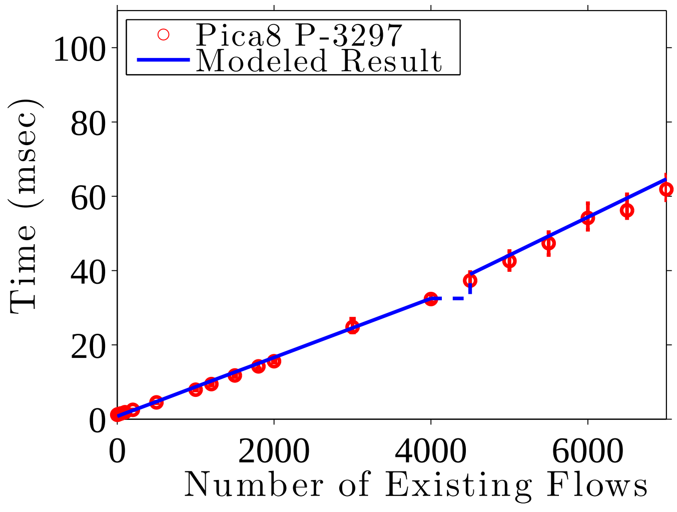
Figure 4.11: Validation results on insertion, modification, and deletion time with descending priority distribution.



(a) Per Flow Insertion Time Comparison.

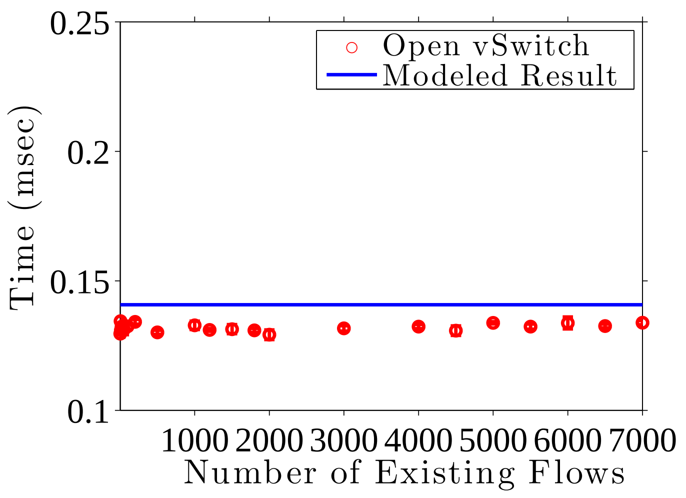


(b) Per Flow Modification Time Comparison.

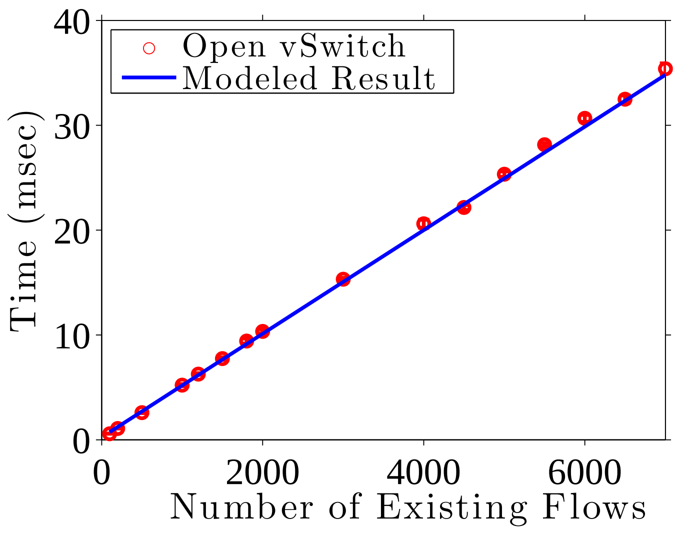


(c) Deletion Time Comparison.

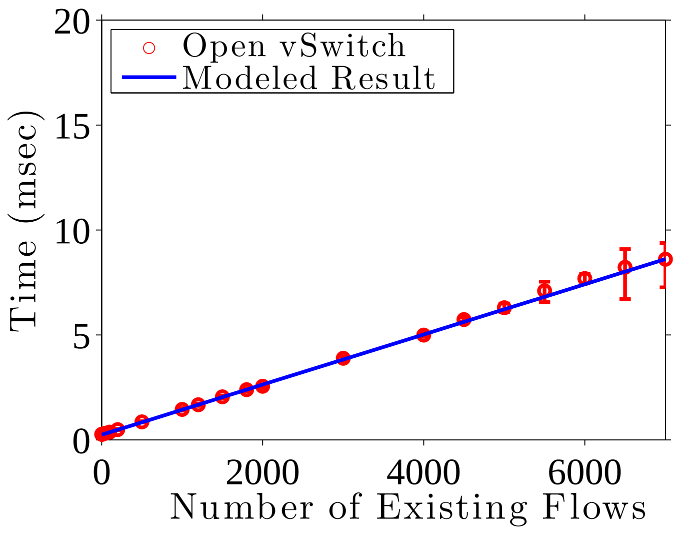
Figure 4.12: Validation results on insertion, modification, and deletion time with same priority distribution.



(a) Per Flow Insertion Time Comparison.

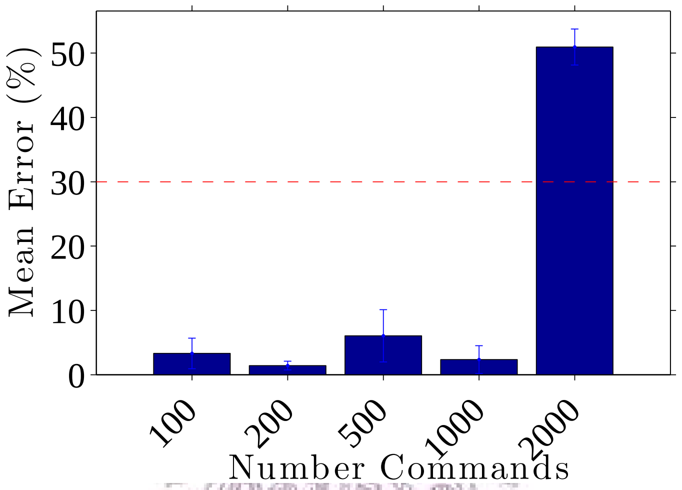


(b) Per Flow Modification Time Comparison.

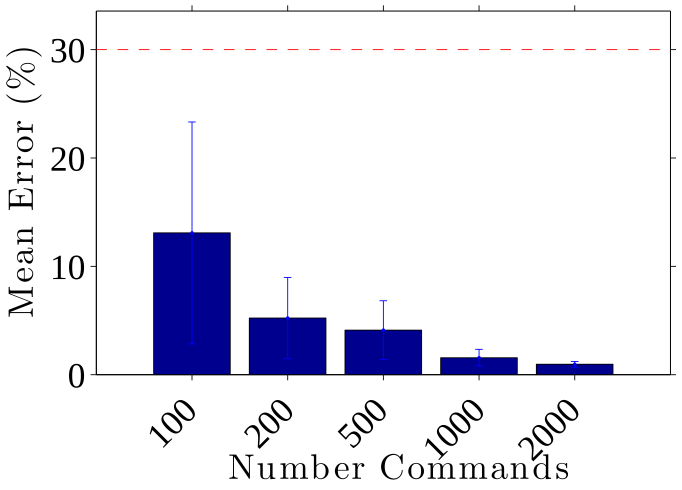


(c) Deletion Time Comparison.

Figure 4.13: Validation results on insertion, modification, and deletion time with ascending priority distribution.



(a) Pica8 P-3297.



(b) Open vSwitch.

Figure 4.14: Validation results on random commands.

Chapter 5

Data Plane Performance Modeling

In terms of data plane performance modeling, we focus on packet forwarding latency and throughput.

5.1 Test Scenarios

We design a series of test scenarios to study on the data plane performances based on the following factors.

- **Matching fields.** We categorize table flows into three groups, flows with matching fields from L2 only, L3 only, and L2 and L3 both. In our experiments, L2 matching fields use ingress port, Ethernet type, and Ethernet destination address. L3 matching fields use ingress port, Ethernet type, and IP destination address. L2 and L3 use both Ethernet destination address and IP destination address for flow entries to match against coming packets.
- **Number of existing flows.** Different flow tables used will result in different implementations on matching data plane packets among the existing flows in the flow table, so the number of existing flows in the flow table may have impact on packet forwarding delay.
- **Inter-packet time.** Data plane packets arrive in different inter-packet time. Data plane traffic may be either overloaded that the switch can be unable to handle in time or underloaded that the switch may queue until there are enough traffic to handle. Either of the conditions may affect the forwarding latency.
- **Packet size.** Dataplane packets are in different packet sizes.

We design a test scenario to measure forwarding delays and throughput using different factors stated above. Fig. 5.1 shows our automatic profiling procedures for data plane

```
1:  let base matching field $H = l23$
2:  let base existing flow size $E = 500$
3:  let base inter-packet time $A = 100$ microseconds
4:  let base packet size $B = 128$ bytes
5:  for each matching field $h = l2, l3, l2l3$ do
6:      for each packet size $b = 64, 128, 256, 512, 1024$ do
7:          insert corresponding flows with $h$ matching fields set and insert other flows
              to fill up to existing flow size of $E$
8:          start packet capturer, tshark, and listen on ingress and egress network inter-
              faces
9:          send data plane packets with packet size of $b$ bytes and inter-packet time of $A$
              microseconds
10:     end for
11: end for
12: for each existing flow size $e = 100, 200, 500, 1000, 1200, 1500, 2000$ do
13:     insert corresponding flows with $h$ matching fields set and insert other flows to fill
          up to existing flow size of $e$
14:     start packet capturer, tshark and listen on ingress and egress interfaces
15:     send data plane packets with packet size of $B$ bytes and inter-packet time of $A$
          microseconds
16: end for
17: for each inter-packet time $a = 100, 200, 500, \ldots, 1000000$ do
18:     insert corresponding flows with $H$ matching fields set and insert other flows to
          fill up to existing flow size of $E$
19:     start packet capturer, tshark and listen on ingress and egress interfaces
20:     send data plane packets with packet size of $B$ bytes and inter-packet time of $a$
          microseconds
21: end for
```

Figure 5.1: The pseudo code of the data plane measurement procedures.

performance measurements. First, we insert a number of flows in the switch flow table. Different matching fields and different number of flows are used for the flows we insert in each of the measurements. Then, we send 100 distinct data plane packets that match the flows in the flow table with same inter-packet time and packet size for all packets. Every test, we use different packet sizes and different inter-packet time. At the same time, we capture packets on both sender and receiver side and calculate the mean latency and throughput from the captured packets.

5.2 Measurement Results

The sample results are collected with L2/L3 matching fields and existing flow size of 500 for flows in the flow table, and inter-packet time of 100 milliseconds and packet size of 128 bytes for data plane packets if not specify, on Pica8 and OvS switches. Similar trends of packet forwarding latency are observed on both Pica8 and OvS switches as shown in Figs. 5.2 and 5.3.

Larger packet sizes result in higher forwarding delays. Switch processing time or the transmission delay can be affected by the packet size. With larger packet size, the processing time and transmission delays increase accordingly for both Pica8 and OvS.

Existing flow sizes have little impact on forwarding delays. Since TCAM is used in Pica8, flow matching can be done quicker and in constant time with different number of existing flows. For OvS, the delay is much higher than that of Pica8 switch since software-implemented switch is used.

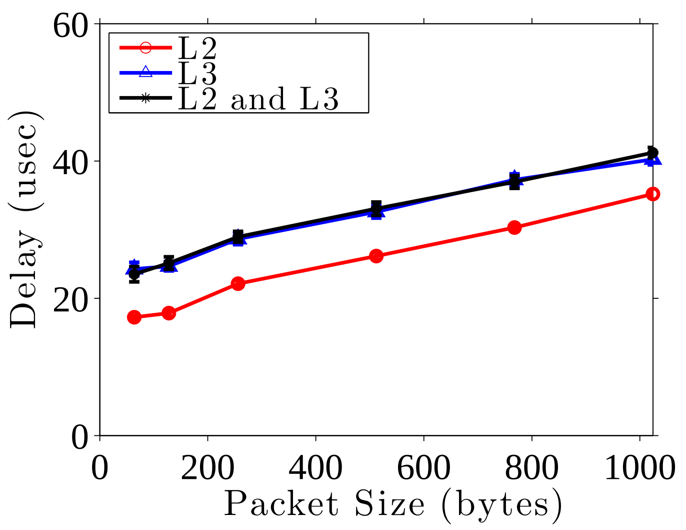
Multi-levels of forwarding delays with different inter-packet time. Delay time is basically constant with different inter-packet time, but it can start to grow when the inter-packet time reaches a certain threshold. We suppose there can be more than two levels, which we observe in the OvS result. In this way, even the results are linearly increasing or decreasing in other switches, our model can still work since linearly increasing/decreasing line can be viewed as multiple levels with only one element in every level.

5.3 Performance Models

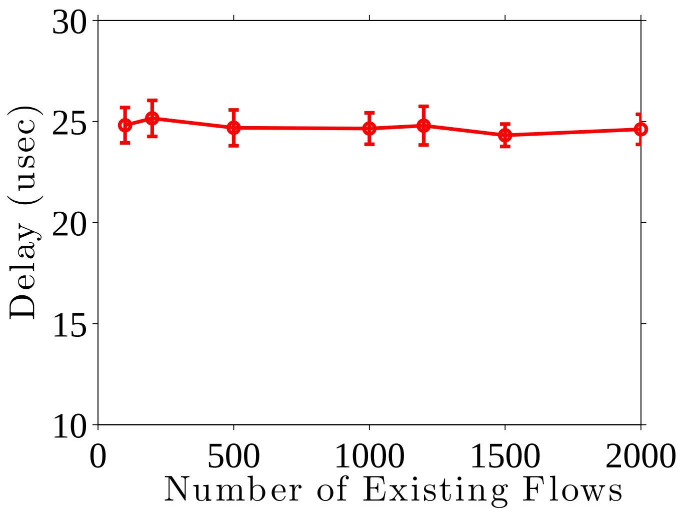
We consider the factors described in Sec. 5.1 as our modeling inputs.

$$D_{delay} = \beta^t_{h_k} + \gamma^t_a \times \Delta a_k + \gamma^t_e \times \Delta e_k + \gamma^t_b \times \Delta b_k \tag{5.1}$$

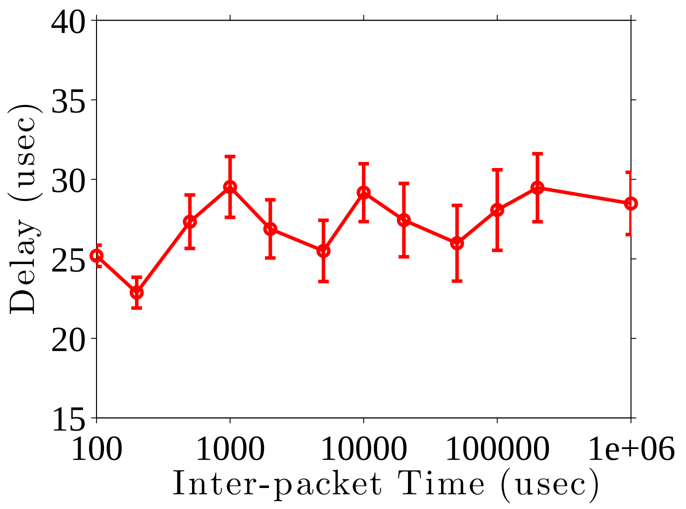
Packet forwarding delay model. Eq. (5.1) shows the model of packet forwarding latency of each incoming packet. β_h, γ_a, γ_e, and γ_b are the switch-dependent parameters,



(a) Sample results on different packet sizes and different matching fields.

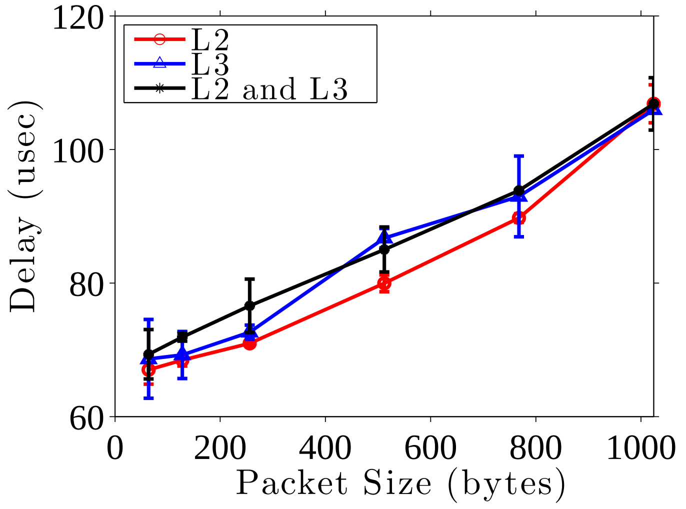


(b) Sample results on different existing flow sizes.

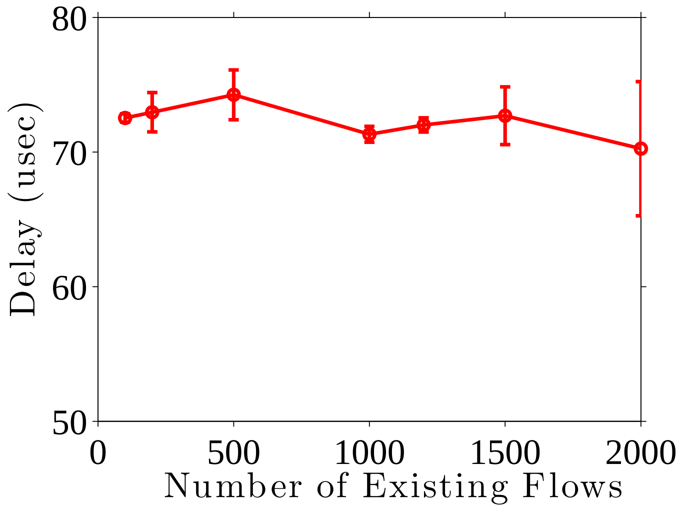


(c) Sample results on different inter-packet times.

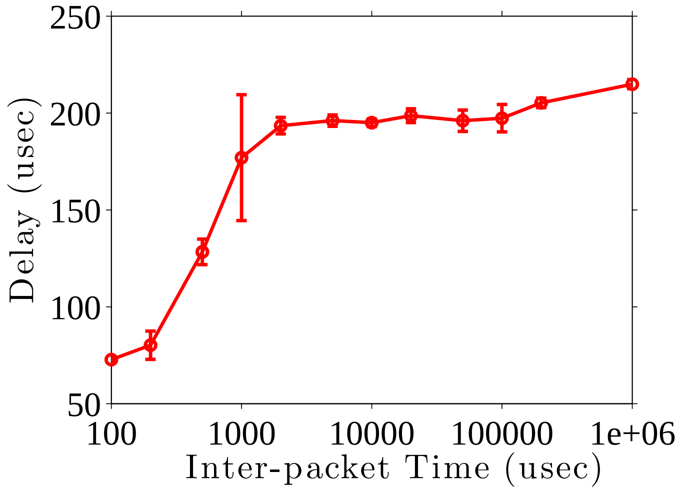
Figure 5.2: Sample results on Pica8 P-3297.



(a) Sample results on different packet sizes and different matching fields.



(b) Sample results on different existing flow sizes.



(c) Sample results on different inter-packet time.

Figure 5.3: Sample results on Open vSwitch

and a_k, e_k, and b_k is the modeling inputs. β_h refers to a base delay time at base inter-packet time A, base existing flow size E, base packet size B, and matching field h. The base inter-packet time A is 100 microseconds, base existing flow size E is 500, and base packet size B is 128 bytes. Base delay time of different matching fields β_h is recorded, respectively. As for existing flow size and packet size, linearly increasing trends are observed in the measurement results of both Pica8 and OvS. A increasing rate is calculated compared to the base delay time and base values of different factors. γ_e^t and γ_b^t refer to the rates for existing flow size and packet size for table t, while Δe_k and Δb_k refer to the differences between existing flow size and packet size to the base values, respectively. For inter-packet time, multi-level of constant delays are observed. We consider the average delay time of each level and do a linear curve fitting. γ_a^t is the increasing rate for different levels for table t, and Δa is the level difference between current inter-packet time a_k and base inter-packet time A.

Switch-dependent parameters.

- **Base delay time, β.** We define the base delay time when packets are sent with inter-packet time of 100 microseconds and packet size of 128 bytes under existing flow size of 500. Delays are increased or decreased when packet size, inter-packet time, or existing flow size changes. The increasing or decreasing rates are also defined as our parameters (described below).

- **Different rates, γ_a, γ_e, and γ_b.** Increasing or decreasing rates for forwarding delay based on the differences between the current states with the base inter-packet time, packet size, or existing flow size values.

- **Number of inter-packet time levels.** As we observe in the measurement study, the results for different inter-packet time are neither constant nor linearly increasing/decreasing, but like stairs, the delay time increases when it reaches a threshold. We allow multiple levels of delays in our models and record the number of levels.

- **Mapping between inter-packet time and inter-packet time levels.** As described above, we assume there can be multiple levels of delay time for different inter-packet time. For each inter-packet time level, we record the inter-packet time threshold when the delays start to change.

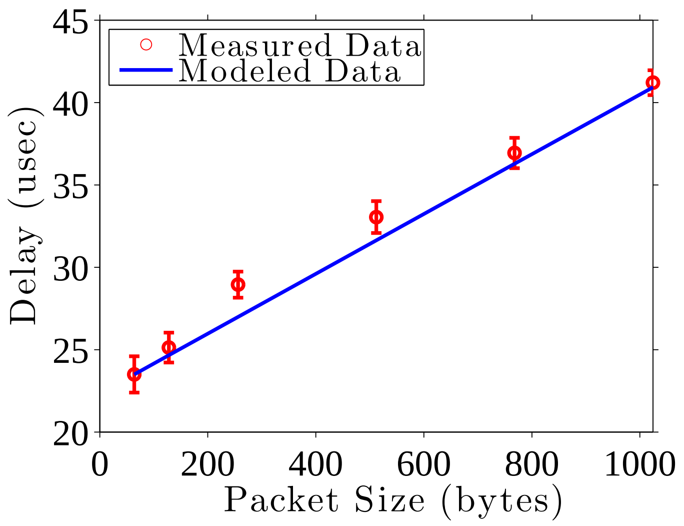
The derivation of every parameters is described as the following. Base delay time, β_{l2}, β_{l3}, β_{l2l3}, is the delay time obtained under the base states we define for three different matching fields. Delays are linearly increasing or constant for different existing flow sizes and packet sizes. Curve fitting with a first-degree polynomial is performed for results of different existing flow size and packet size to the forwarding delays, respectively. The first-degree term for both results are γ_e and γ_b.

For number of inter-packet time levels and mappings between inter-packet time and inter-packet time levels, we examine the forwarding delays under different inter-packet time. For any two consecutive inter-packet time samples, we check if the forwarding delays are close enough to be put in the same level. We calculate delay ranges with upper bound equal to the forwarding delay plus the standard deviation of the forwarding delay and lower bound equal to the delay minus the standard deviation. Obtaining delay ranges for two inter-packet time samples, we check if the range is overlapped. If they are overlapping, we consider both samples are in the same level; if not, they are different levels. We examine through all inter-packet time samples and generate a mapping between inter-packet time and inter-packet time levels by recording the threshold of each inter-packet time level. The average forwarding delays are calculated for each level, and curve fitting of a first-degree polynomial is performed on the delays of every level. Then, the first-degree term is the increasing rate for different inter-packet time levels, γ_a.

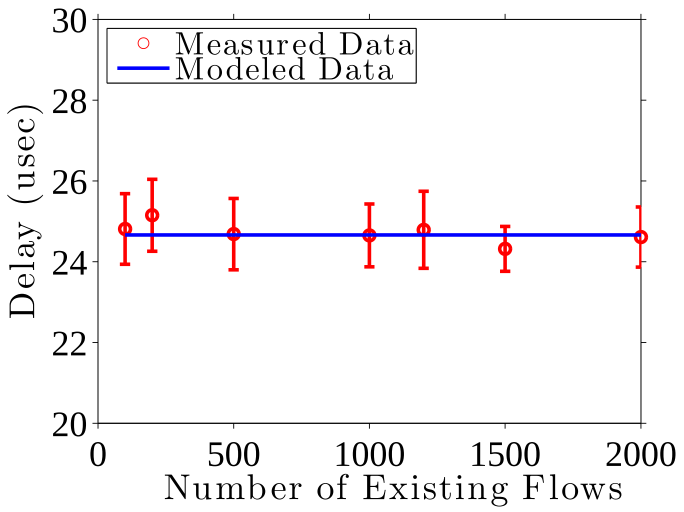
5.4 Model Validation

We validate our modeling results in two test scenarios. Firstly, we use the same scenarios as we use for our profiling measurements. Different matching fields, existing flow size, packet size, and inter-packet time are used. Performance metrics of latency and throughput are compared against the ground truth in our validation results. Figs. 5.4, 5.5, 5.6, and 5.7 show the results for packet forwarding delay and throughput on Pica8 P-3297 and Open vSwitch, respectively. The modeling results follow the measured results closely on both Pica8 and OvS.

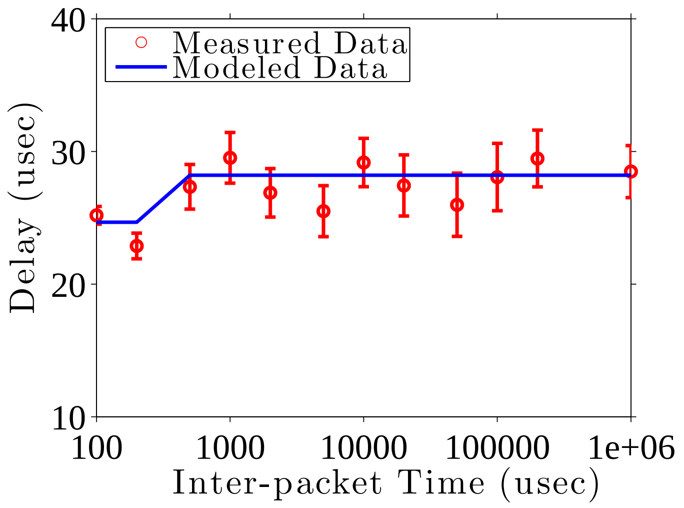
We also validate our model under a random test scenario. The data plane packets are from a real world trace [7], which is captured from a small educational organization with about 35 employees and 100 students working and studying at the site. 100 Mbps for LAN connection in this location and 1 Gbps for core networks are used. Over 2000000 packets are captured in this real world trace. The traffic over the measured link is mildly loaded during the measurement period, May to June in 2007. Our validation experiments test under different number of data plane packets sent, and they are $\{100, 200, 500, 800, 1000, 2000\}$. We randomly select packets from this trace. For each number of packet, we randomly select 16 different traces for the experiments. We compare the modeling results by calculating the error rates for each number of packet. Error rate is given by the differences between the modeling results and the ground truth divided by the ground truth. Average values and 95% confidence intervals are calculated from



(a) Different packet sizes.

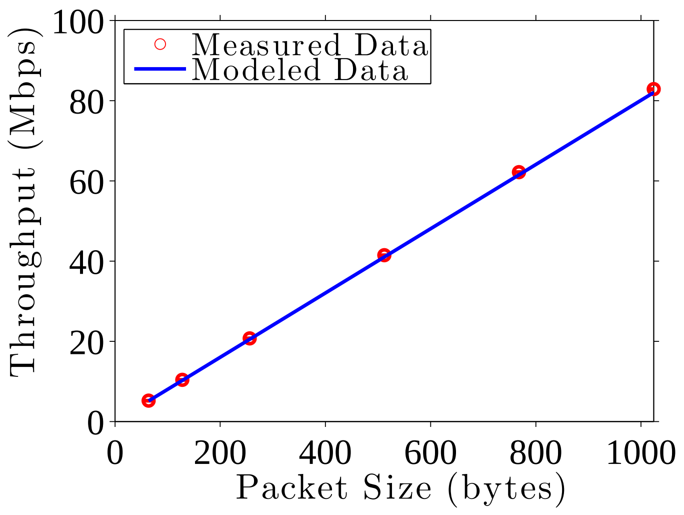


(b) Different existing flow sizes.

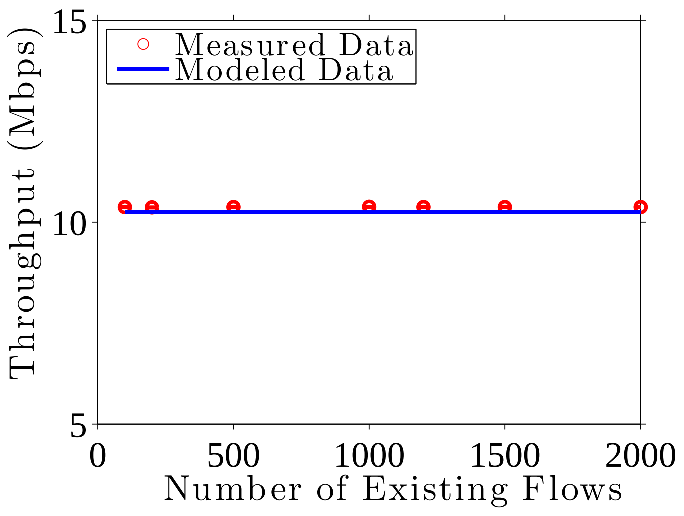


(c) Different inter-packet time.

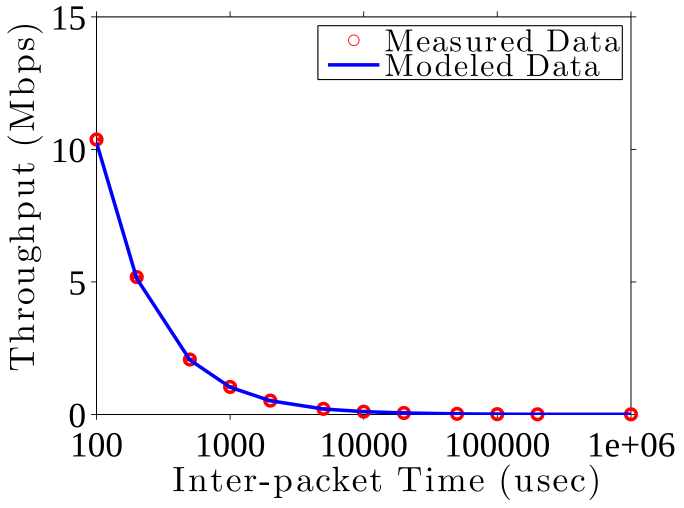
Figure 5.4: Validation results on delays with different factors on Pica8 P-3297.



(a) Different packet sizes.

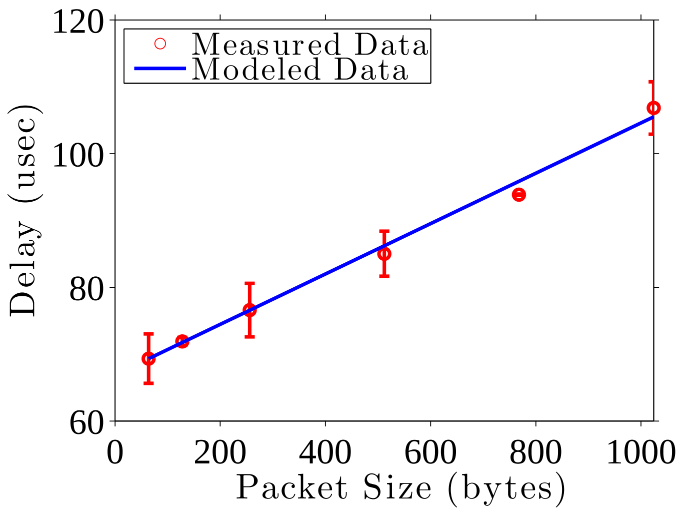


(b) Different existing flow sizes.

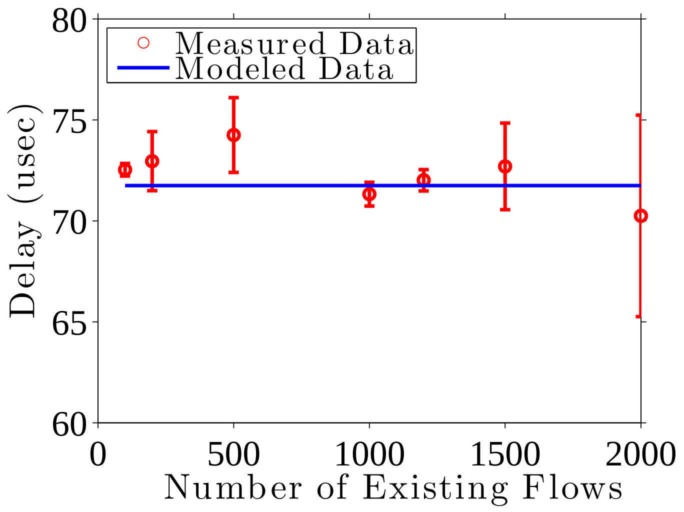


(c) Different inter-packet time.

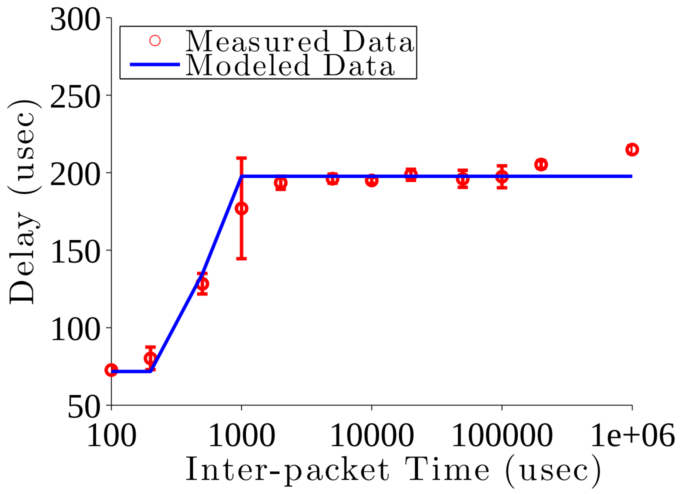
Figure 5.5: Validation results on throughputs with different factors on Pica8 P-3297.



(a) Different packet sizes.

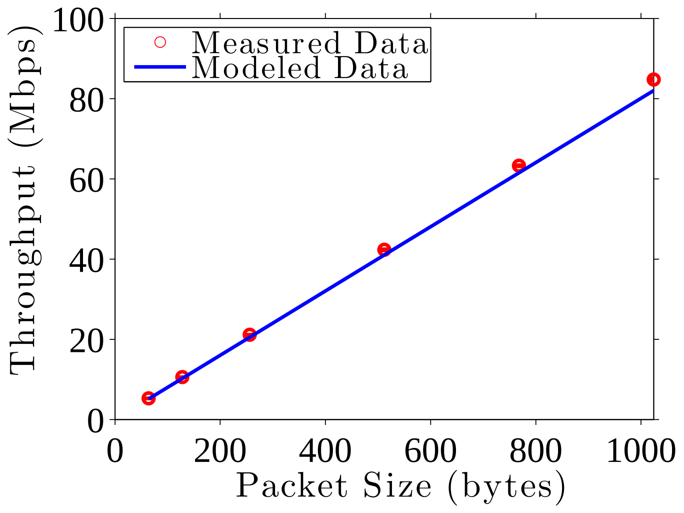


(b) Different existing flow sizes.

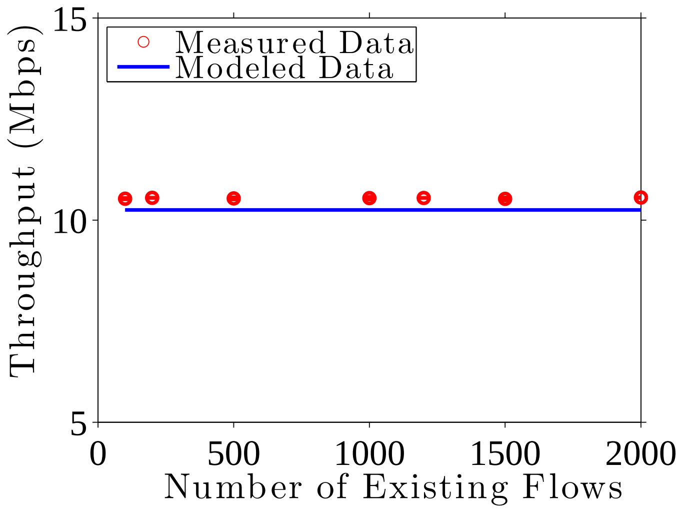


(c) Different inter-packet time.

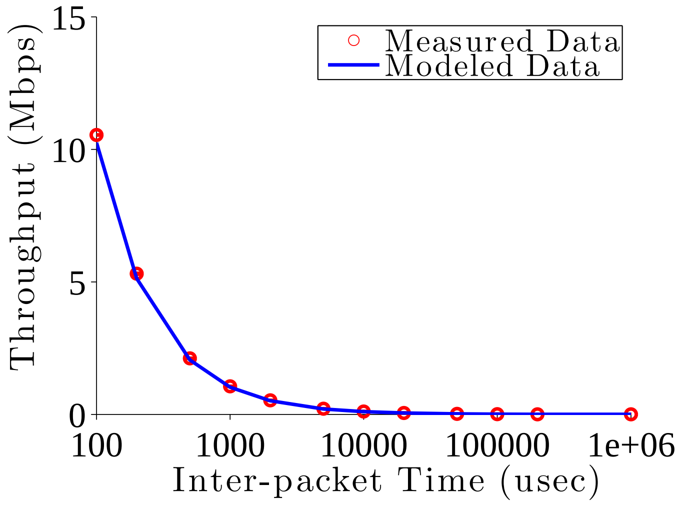
Figure 5.6: Validation results on delays with different factors on Open vSwitch.



(a) Different packet sizes.

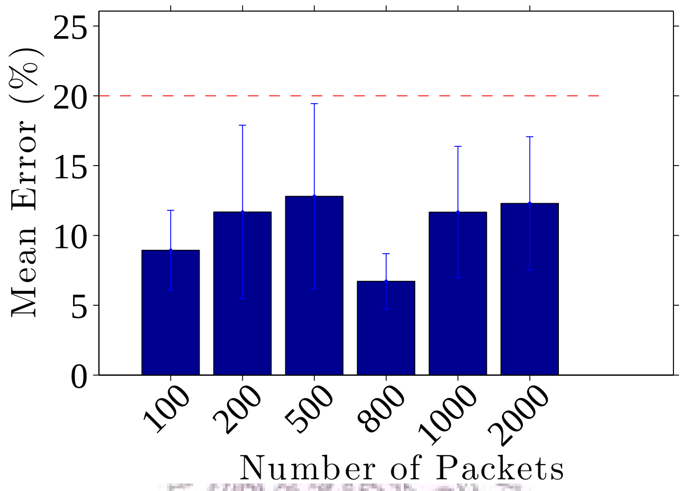


(b) Different existing flow sizes.

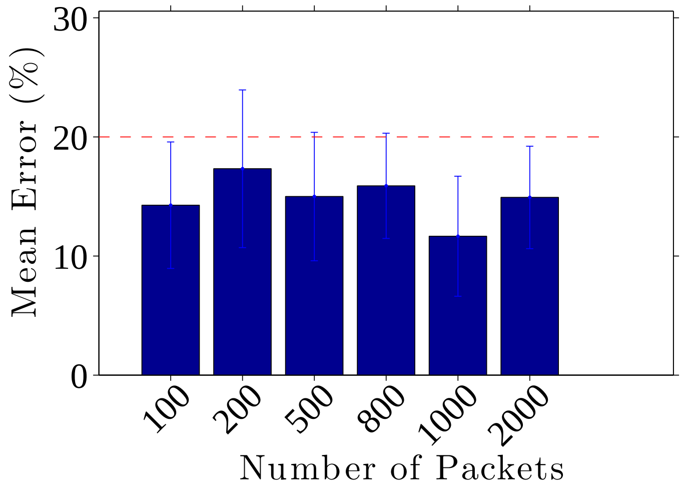


(c) Different inter-packet time.

Figure 5.7: Validation results on throughputs with different factors on Open vSwitch.



(a) Pica8 P-3297.



(b) Open vSwitch.

Figure 5.8: Validation results using a real world trace.

the 16 samples for each number of packet. Fig. 5.8 shows the validation result for Pica8 P-3297 and Open vSwitch, respectively.

Chapter 6

Emulator Implementations and Evaluations

The essential criteria of our emulator design is to gather switch state information and integrate performance models that leverage the state information as input to achieve better performance accuracy. As a result, the control plane performance can conform to a real OpenFlow switch.

6.1 Design

Fig. 6.1 shows the design of our detailed emulator. To better emulate control plane performance, the first key point is to detect and capture *flow_mod* events from those controller/switches messages. Therefore, when an OpenFlow controller sends messages to the switch via a dedicated control channel, an *OpenFlow Event Detector* examines the incoming messages and extracts OpenFlow *flow_mod* events from them, and *flow_mod* events are put into a *OpenFlow Event Queue* for further event handling process. Subsequent OpenFlow events sometimes come in with relatively shorter intervals, so a burst data transmission may cause multiple OpenFlow events to wait. We suppose there is optimization technique implemented to leverage this characteristics for more efficient processing. So the design of a event queue is to emulate multiple OpenFlow events come and wait for batch processing at the same time.

An *OpenFlow Event Handler* fetches events from the event queue and processes the events accordingly, including flow insertions, flow modifications, and flow deletions (which already implemented in OvS). The event information is passed to *Clock Time Manager*, which adjusts the time by adding delays based on the event information, the parameters specified in the configuration file, and switch state information. The event information is at the same time passed to a *Switch State Maintainer*, which monitors the

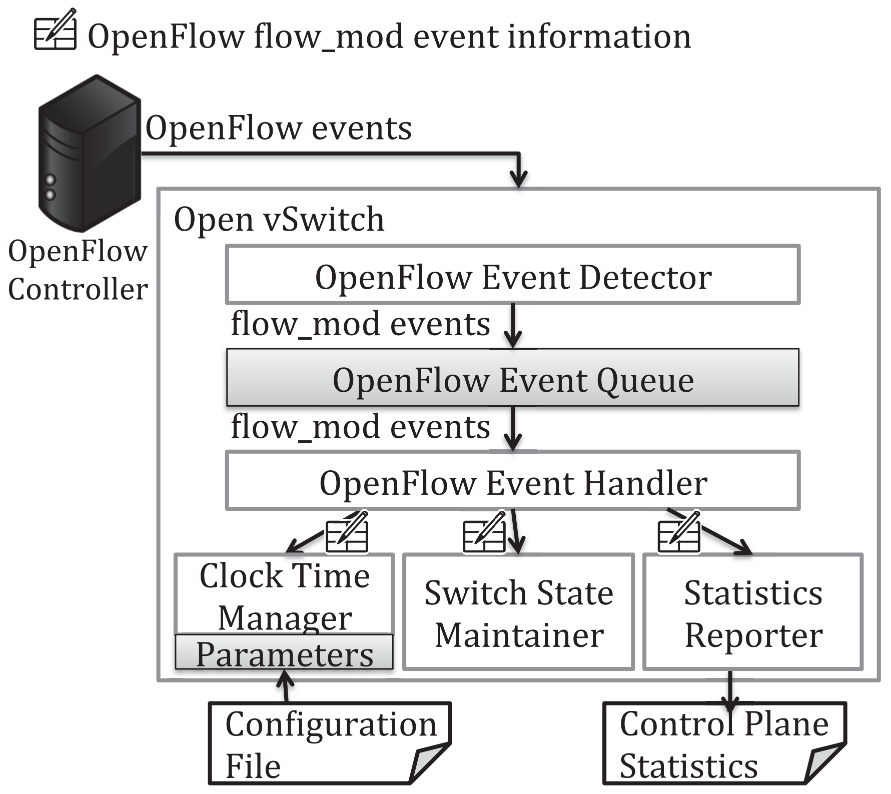


Figure 6.1: System Design.

switch states, such as number of flows in the flow table and the priority distribution of existing flows, so that we can do accurate modeling using those information for clock time manager to adjust the clock time. After the clock time adjustment for better performance accuracy, the event information is then passed down to a *Statistics Reporter*. The statistics reporter is responsible to record every control plane event information and the processing time of each event into a file.

6.2 Implementation

In this section, we will detailed describe how each component in Fig. 6.1 is implemented.

Parameter configuration. Each time when an OvS in Mininet is created and initialized, the parameter configuration file is read and the corresponding parameters are initialized by the values written in the configuration file. The configuration file is written as a *xml* file. Each parameter is declared within the corresponding xml tag. In OvS, we compile with opensource library *libxml2* and leverage libxml2 API to parse configuration files and obtain switch-dependent parameters.

OpenFlow event queue. In our implementation, we mainly focus on the flow_mod messages such as flow insertion, modification, and deletion time, so we implement a event queue that can buffer *flow_mod* events from the controller. We implement a circular queue to hold *flow_mod* events. In the original design of OvS, it treats all kinds of OpenFlow events the same way before it goes down to next steps (messages processing). We especially extract *flow_mod* events from the very beginning and puts them into our event queue. The event queue is in a first-come-first-serve (FCFS) sense. So the *flow_mod* event handler fetches the oldest events from the event queue and follows the originally-implemented flow_mod process in OvS. Since two handlers (OpenFlow event detector and handler) access the same event queue at the same time, we keep two variables to indicate where we should put events and where we could get events from the queue. The design of the event queue is to emulate the batch processing in switch implementations. Every event arrive before time up will be saved in the same batch (event queue) at the same time.

Switch state maintainer. Each event may involve modifications to the flow table. The responsibility for switch state maintainer is to extract necessary data and update the switch states accordingly. We mainly maintain two values of switch states. One is the number of existing flows, and the other is the priority distribution of existing flows. The event contains information about which type of *flow_mod* operation is used now. For flow insertion events, each time only one flow can be added, so the existing flow size is increased by 1 when a flow insertion event is detected. For flow modification events, it does not change the existing flow size, so the value remains unchanged. For deletion events, there may be two conditions. If the flow is strictly matched, only one flow will be involved; otherwise, if wildcarded matching is used, we should examine the value that indicates number of flows in the table matched to the coming command, and the number of existing flows should decrement accordingly.

For priority distribution, since the priority value ranges from 0 to 65535, so we define an array with size 65536, each maintains the number of flows using this priority value. For instance, if we want to know how many flows are using priority 5, we simply obtain the value with index 5, array[5]. Any insertion and deletion event will change the priority distribution. Each time a flow insertion is detected, the specified priority is extracted and the corresponding number is updated. For deletion event, every matching flow should be examined and decrement the number accordingly.

Clock time manager. When the switch is created and initialized, the parameters are also read from file and initialized. With switch-dependent parameters, event information, and switch states maintained by switch state maintainer, we can calculate the delays using our performance models and place delays accordingly. For all three types of commands, we need to examine the priority distribution first to see if single or multiple priorities is used, and then choose the performance models based on the flow_mod command type.

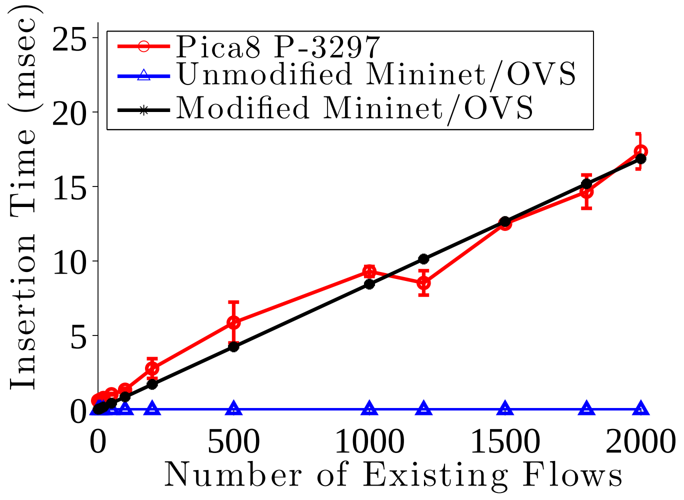
Control plane statistics reporter. After processing by clock time manager, the *flow_mod* event information is passed down. We record the command type, such as *add*, *modify*, *delete*, and *idle timeout delete*. The starting timestamp of the command records the time we receive this command, and delay indicates the processing time of the command.

6.3 Evaluations

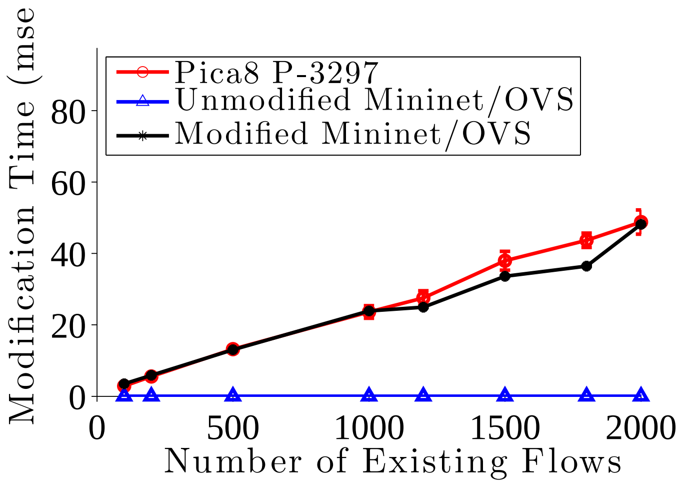
We have conducted the same series of test scenarios we define in Ch. 3, flow insertions, flow modifications, and flow deletions.

We setup the OFLOPS controller on a Ubuntu desktop with Intel i7-4790 3.6 GHz 8-core CPU and 8 GB memory. We use OFLOPS testing modules to measure performances for three different testbeds: Pica8 P-3297, Mininet, and modified Mininet. Pica8 P-3297 testbed setup is the same as we describe in Ch. 3. For Mininet and our modified Mininet, we setup the emulator on the same PC that runs the OFLOPS controller. We instantiate a network topology of one switch with two hosts connected for both testbed setup.

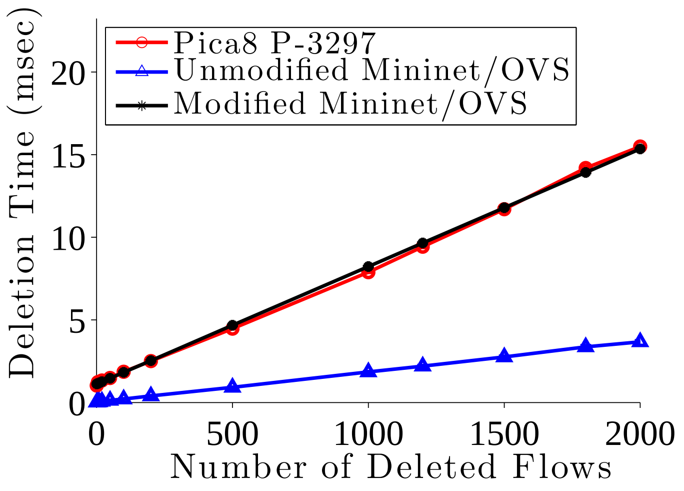
Fig. 6.2 show the evaluation results for flow insertion, flow modification, and flow deletion scenarios. Original Mininet use a software-based flow table, and the flow table is implemented using hashtable-based design so that flow table modification whatever insertion, modification, or deletion OpenFlow event occur under different switch state, the time cost is relatively small due to less computation and higher power of CPU. So the Mininet/OvS results are far differently from the Pica8 switch. While for our detailed emulator, by inserting delays based on the model and parameters we derive from measurement study, the performance is quite close to the result from Pica8 under three test scenarios.



(a) Flow insertion time.



(b) Flow modification time.



(c) Flow deletion time.

Figure 6.2: Evaluation results of three test scenarios.

Chapter 7

Conclusion

We conducted extensive measurement studies on hardware and software-implemented OpenFlow switches in terms of control plane and data plane performances. *Switch performance models* were proposed in order to accurately emulate OpenFlow switch performances. Our models take inputs of switch states and several configurable *switch-dependent parameters*. By adjusting the parameters, the performance models are able to emulate different switch implementations from different vendors. The switch-dependent parameters can be generated from our proposed *automatic switch profiling procedures*, which were derived from the measurement studies we conducted. Each switch has its own set of switch-dependent parameters. We also conducted several validation experiments to validate accuracy of our performance models. We generated random commands with match fields set in random-generated values for control plane performances validation. The error rates were mostly under 30% for both Pica8 and OvS. For data plane performance accuracy validation, we used a real world *PCAP trace* and randomly selected a series of packets from it. The error rates were all under 20% in our experiments on Pica8 and OvS.

We also integrated our performance models into an existing OpenFlow emulator, Mininet/OvS. We conducted several experiments to examine the emulator implementations. With the integration of our performance models, modified Mininet/OvS had great improvements in control plane performance emulations compared to the original Mininet implementations and was closely following the performance results from the switch we managed to emulate.

7.1 Future Directions

Following are several directions to work on.

Deep insight into the switch implementations. Several measurement results reveal interesting observations that we can further look into in details. Control plane performances, such as insertion and modification delays, reveal that batch processing for a bunch of commands may be present, and performances can take benefits from optimizations of command batch processing. Also in some control plane performance measurement results, the outcome is not intuitive to the best of our knowledge, such as linearly increasing delays for insertion command processing with same priority distribution or modification command processing under different numbers of existing flows. In data plane performance measurement results with different packet inter-arrival time, the results are almost constant under a certain packet rate. The presence of queues or other implementations of switch data plane should be detailed examined. Above all, the hardware switch implementations are more sophisticated than we thought of. To further understand how the underlying switch works, design of series of experiments and studies of extensive measurement results can be conducted. Open vSwitch group newly released a paper [32] on its switch architecture and implementations. Though switch implementations differ, they should still share similarly on the high level design and ultimate goal, and thus the implementation details on data plane and flow table can still give ideas to the design of experiments.

Packet forwarding delays should be excluded from control plane performances. For control plane performance measurements, the flow completion is notified by the reception of corresponding data plane packets. Though the forwarding delays are relatively small compared to the control plane performances, it should be excluded for more accurate measurement results. It can be done by modifying OFLOPS testing modules. In addition to recording the starting time of *flow_mod* events and flow completion time, sending and receiving timestamps of notification-use data plane packets should be recorded by *Packet Handler* in order to take packet forwarding delays into considerations.

Accuracy of control and data plane performance modeling. From above, more measurements can be conducted to study in details on the switch architecture. More observations and information of switches can be obtained. Also the proposed performance models still have room for improvements if more general conditions of network traffic are considered. Performance models can be improved from a further study on switch architecture.

Complete data plane measurements, implementations, and evaluations. Data plane performance measurements on software table need conducting to complete the overall data plane performance measurements on Pica8 switch. In addition, currently our emulator design is based on control plane performance. The data plane performance model has not yet been integrated into Mininet/OvS. With data plane implementations, more thorough and general performance accuracy evaluations of the detailed emulator can also be conducted, for instance, real world traces can be used to evaluate the correctness of the emulator.

Update OFLOPS to OpenFlow 1.3 compatible. Currently, our automatic switch profiling procedures are based on the open source project, OFLOPS. However, OFLOPS is implemented based on OpenFlow specification 1.0.0, but several available OpenFlow switches are upgraded to newer version of OpenFlow, though some switches are backward compatible, there are still others only compatible with OpenFlow 1.3 or any newer version. For switch performance measurements and more switch-dependent parameters derivation, update OFLOPS to a newer version should be necessary.

Bibliography

[1] Engineered elephant flows for boosting application performance in large-scale clos networks. http://zh-tw.broadcom.com/collateral/wp/OF-DPA-WP102-R.pdf.

[2] Mininet: An instant virtual network on your laptop. http://http://mininet.org/.

[3] ns-3: OpenFlow switch support. http://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html/.

[4] Oflops. http://archive.openflow.org/wk/index.php/Oflops.

[5] Open vswitch: Production quality, multilayer open virtual switch. http://openvswitch.org/.

[6] OpenFlow switch specification 1.4.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[7] Pcap traces. http://www.simpleweb.org/wiki/Traces#Pcap_Traces.

[8] Tcpdump and libpcap. http://www.tcpdump.org.

[9] Tcpreplay: Pcap editing and replay tools for *nix. http://tcpreplay.synfin.net.

[10] tshark-dump and analyze network traffic. https://www.wireshark.org/docs/man-pages/tshark.html.

[11] 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 *IEEE Global Communications Conference (GLOBECOM'13)*, 2013.

[12] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of the 10th ACM SIGCOMM conference on Internet measurement (IMC'10)*, 2010.

[13] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. OpenFlow switching: Data plane performance. In *IEEE International Conference on Communications (ICC'10)*, 2010.

[14] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *IEEE International Conference on Cloud Networking (CloudNet'14)*, 2014.

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

[16] M. P. Fernandez. Comparing OpenFlow controller paradigms scalability: Reactive and proactive. In *IEEE 27th International Conference on Advanced Information Networking and Applications (AINA'13)*, 2013.

[17] A. Gelberger, N. Yemini, and R. Giladi. Performance analysis of software-defined networking (SDN). In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'13)*, 2013.

[18] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for SDN prototyping. In *Proc. of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, 2013.

[19] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. of the 8th International conference on emerging networking experiments and technologies (CONEXT'12)*, 2012.

[20] B. Heller. *Reproducible network research with high-fidelity emulation*. PhD thesis, Stanford University, 2013.

[21] T. R. Henderson, M. Lacage, and G. F. Riley. Network simulations with the ns-3 simulator. In *Sigcomm Demo*, 2008.

[22] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, 2013.

[23] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an OpenFlow architecture. In *Proc. of the 23rd International Teletraffic Congress (ITC'11)*, 2011.

[24] X. Kong, Z. Wang, X. Shi, X. Yin, and D. Li. Performance evaluation of software-defined networking with real-life ISP traffic. In *IEEE Symposium on Computers and Communications (ISCC'13)*, 2013.

[25] D. Kreutz, F. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, January 2015.

[26] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX'10)*, 2010.

[27] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization. In *Proc. of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CONEXT'14)*, 2014.

[28] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel. On the modeling of OpenFlow-based SDNs: The single node case. In *Proc. of Computer Science and Information Technology (CS&IT'14)*, 2014.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[30] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys and Tutorials*, 16(3):1617–634, February 2014.

[31] R. Olsson. Pktgen the linux packet generator. In *Proc. of the Linux Symposium*, 2005.

[32] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *Proc. of 12th USENIX Symposium on Networked Systems Design and Implementation (USENIX NSDI'15)*, 2015.

[33] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An open framework for openflow switch evaluation. In *Proc. of the 13th International Conference on Passive and Active Measurement (PAM'12)*, 2012.

[34] M. Shibuya, A. Tachibana, and T. Hasegawa. Efficient performance diagnosis in OpenFlow networks based on active measurements. In *Proc. of the 13th International Conference on Networks (ICN'14)*, 2014.

[35] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield. Efficient network-wide flow record generation. In *Proc. IEEE INFOCOM*, 2011.

[36] S. Wang. Comparison of SDN OpenFlow network simulator and emulators: EstiNet vs. Mininet. In *IEEE Symposium on Computers and Communication (ISCC'14)*, 2014.

[37] S. Wang, C. Chou, and C. Yang. EstiNet openflow network simulator and emulator. *IEEE Communication Magazine*, 51(9):110–117, September 2013.

[38] Wireshark official site, August 2010. `http://www.wireshark.org`.

[39] W. Xia, Y. Wen, C. Foh, D. Niyato, and H. Xie. A survey on software-defined networking. *IEEE Communications Surveys and Tutorials*, 17(1):27–51, March 2014.

[40] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of Software-Defined Networking. *IEEE Communication Magazine*, 51(2):136–141, 2013.