ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

CONGESTION AND PACKET CLASSIFICATION BASED FLOW MANAGEMENT FOR SOFTWARE-DEFINED NETWORKS

M.Sc. THESIS

Mertkan AKKOÇ

Department of Computer Engineering

Computer Engineering Programme

July 2020

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

CONGESTION AND PACKET CLASSIFICATION BASED FLOW MANAGEMENT FOR SOFTWARE-DEFINED NETWORKS

M.Sc. THESIS

Mertkan AKKOÇ
(504171519)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. Berk CANBERK

July 2020

ISTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

YAZILIM TANIMLI AĞLARDA TIKANIKLIK VE PAKET SINIFLANDIRMA ODAKLI AKIŞ YÖNETİMİ

YÜKSEK LİSANS TEZİ

Mertkan AKKOÇ
(504171519)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Assoc. Prof. Dr. Berk CANBERK

Temmuz 2020

Mertkan AKKOÇ, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 504171519 successfully defended the thesis entitled "CONGESTION AND PACKET CLASSIFICATION BASED FLOW MANAGEMENT FOR SOFTWARE-DEFINED NETWORKS", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Berk CANBERK** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Güneş Zeynep KARABULUT KURT** Istanbul Technical University	.....
	Prof. Dr. Suat ÖZDEMİR Hacettepe University	.....

Date of Submission : 11 June 2020
Date of Defense : 14 July 2020

To my family,

FOREWORD

As someone who chases the new, likes to do new jobs, develops, while looking for a subject to work in my master's degree, Software Defined Networking (SDN) has caught my attention. When I started my master's degree, I did not have detailed information about SDN. However, the information I had was enough to excite me. Over time, I learned more detailed information by researching. The benefits of SDN technology, what they can do, cannot do, problems of it came out one by one. Among all these problems, I chose two problems: congestion of SDN controller under heavy traffic and insufficiency of package classification in OpenFlow vSwitch under the increase in the number of rule fields. These two problems waiting to be solved are the reasons for the emergence of this thesis.

Firstly, I would like to thank my advisor, Assoc. Prof. Dr. Berk CANBERK from Computer Engineering Department of Istanbul Technical University, who advises me not only in my studies but also in my life journey. Maybe my graduate life, this thesis would not exist if there isn't his patience towards me, his belief in my ability to do. He conveyed this belief to me every time. Also, I would like to thank the BCRG family of his research group for accepting me, helping me tirelessly, being ready to help, and always being friendly.

I would like to extend my thanks to all of my friends, who I laughed together, had fun, got upset, are not only friends but also a family for me. I could not separate one from the other. Maybe I would have been in a completely different place and situation if they had not accepted me. Finally, I would like to thank my family, who is my permanent and unconditional shelter, for their patience and support.

This thesis was supported by ITU Scientific Research Fund with a project number: 42439. Also, this thesis was supported by the Turkcell-Istanbul Technical University Researcher Funding Program.

June 2020 Mertkan AKKOÇ

TABLE OF CONTENTS

Page

ABBREVIATIONS

ACL **:** Access Control Lists
API **:** Application Programming Interface
BCAM **:** Binanrt Content Addressable Memory
CAM **:** Content Addressable Memory
CapEx **:** Capital Expenditure
CPU **:** Central Processing Unit
eMBB **:** Enhance Mobile Broad
FPGA **:** Field Programmable Gate Array
FW **:** Firewalls
IP **:** Internet Protocol
IPC **:** IP chains
LRED **:** Loss Ratio-Based RED
mMTC **:** Massive Machine-Type Communication
NFV **:** Network Function Virtualization
ONF **:** Open Networking Foundation
ONOS **:** Open Network Operating System
OpEx **:** Operational Expenses
QoC **:** Quality of Controller
QoS **:** Quality of Service
RAM **:** Random Access Memory
RED **:** Random Early Detection
SDN **:** Software-Defined Networking
SDONs **:** Software Defined Optical Networks
TCAM **:** Ternary Content Addressable Memory
TSS **:** Tuple Space Search
T-SDN **:** Transport-Software Defined Network
UAV **:** Unmanned Aerial Vehicles
URLLC **:** Ultra-Reliable and Low-Latency Communications
UWSNs **:** Underwater Wireless Sensor Networks
1D **:** One Dimension

2D **:** Two Dimension

5G **:** The Fifth Generation Mobile Networks

SYMBOLS

f_j	**:** Drop ratio for each flow type
f^*_j	**:** The expected drop rate for each flow type
l_j	**:** Queue length for each flow type in admission management step
l_{sj}	**:** Steady-state queue length for each flow type in admission management step
m_{nj}	**:** Total number of rejected packet_in messages for each flow type
m_{aj}	**:** Total number of incoming packet_in messages for each flow type
N	**:** The length of each queue in priority management step
P	**:** The priority value for a node
p_j	**:** The priority value for each flow type
r_j	**:** Drop probability for each flow type
s	**:** Shows whether a queue exceeds a threshold value or not
s_i	**:** Intervals of rules in 1D
t	**:** PACKET_IN message goes from admission step to prioritization
w_j	**:** Weighting factor for each flow type
γ	**:** Constant value

LIST OF TABLES

Page

LIST OF FIGURES

Page

CONGESTION AND PACKET CLASSIFICATION BASED FLOW MANAGEMENT FOR SOFTWARE-DEFINED NETWORKS

SUMMARY

In this thesis, we focus on problems in the control plane and problems in the data plane of SDN separaterly. In the control plane, we specifically try to increase the response time of the SDN controller in ultra-dense scenarios. In the data plane, we aim to construct an efficient data structure to achieve both fast rule update and fast packet classification.

In the SDN, the control plane is responsible for deciding route and operations for flows that coming to the data plane. To do so, the SDN controller in the control plane has a central view and controls all switches in the data plane. But, this can cause an increase in both e2e latencies of packets and drop rate in the controller if there is a high spiky demand of incoming heterogeneous flows. Because, switches in data plane have to ask what to do to the controller if there is a new incoming flow to them. When newly coming flows increase, communication traffic between the controller and data plane increase. As a result, this can cause congestion in the SDN controller, and e2e latency and drop rate in the controller increase because of this congestion. To solve these problems, we propose a management engine to implement in the SDN controller in ultra-dense SDN scenarios. In this engine, we propose two steps: admission and prioritization steps. We also create different queues for different types of 5G flows (URLLC, eMBB, mMTC) in each step. In the admission, we modify Loss Ratio-Based Random Early Detection (LRED) Algorithm. In prioritization, we propose a tree-based prioritization that considers the priority needs of different flow types and near future states of different queues. According to simulation results, our response time of the SDN controller, e2e latency of packets and dropped rate in the controller are better up to 53%, 58%, and 36%, respectively.

Packet classification is a key factor for choosing proper action for incoming packet and has to be done fast, especially in OpenFlow. But OpenFlow vSwitch technology doesn't allow to use some fast hardware technology for packet classification like TCAM. Decision tree methods are preferred solutions for fast classification in OpenFlow vSwitch in the literature. But most of these methods can cause the rule replication problem. As a result, while the duration of packet classification decreases, rule update duration increases. There are also rule partitioning methods in the literature to solve this problem, but the running time of these methods mostly depends on the number of rule fields. Also, some of these solutions don't overcome the rule replication problem. At that point, the main question is that how can we make the rule partitioning fast by both preventing the rule replication and allowing fast packet classification and rule update in OpenFlow vSwitch? To solve the rule partitioning problem, we convert this problem to the interval partitioning and propose a classic Greedy Algorithm. As a result, the running time of the partitioning algorithm only depends on the rule number. After partitioning, we propose to use HyperCuts to construct decision trees for fast packet classification and rule update. According to performance evaluation results, we

do the rule partitioning and rule updates faster than the PartitionSort method with the percentage of 88, 15, respectively. We also classify packets faster than the TupleMerge method with the percentage of 40 for online and 50 for offline scenarios.

YAZILIM TANIMLI AĞLARDA TIKANIKLIK VE PAKET SINIFLANDIRMA ODAKLI AKIŞ YÖNETİMI

ÖZET

Yazılım Tanımlı Ağlar, klasik ağlardan farklı olarak kontrol düzlemi ve veri düzlemi olarak iki katmana ayrılmıştır. Kontrol düzlemi ağa gelen akışlara hangi işlemlerin uygulanacağı, hangi yollardan gönderilmesi gerektiği kararlarını verip veri düzlemine iletmekle yükümlüdür. Veri düzlemi ise gelen akışların paketlerini, içerisinde var olan kurallar ile karşılaştırıp uygun kuralı bulduktan sonra pakete ilgili işlemi uygulamakla yükümlüdür. Kendi içerisinde uygun kuralı bulamaz ise ilgili akış için hangi kuralların uygulanması gerektiğini kontrol düzlemine sormaktadır. İki düzlem de kendi içerisinde ayrı ayrı ve birbirinin işleyişini etkileyen sorunlara sahiptir. Bu tezde hem kontrol düzlemindeki hem de veri düzlemindeki sorunlara odaklanılmıştır.

Yazılım Tanımlı Ağ teknolojisinde, kontrol düzlemi, ağa gelen verilere uygulanacak işlemleri karar vermesinden dolayı, tüm ağdan sorumludur. Kontrol düzleminde gerekli kararları alan cihaz kontrolör olarak isimlendirilmekte, kontrol düzleminin yapması gereken işlemlerden dolayı tüm ağa hakim olacak merkezi bir konuma sahiptir. Ancak bu merkezi konum ve her şeyden sorumlu olmak, veri düzleminde trafik artışı olduğunda işleyişini etkilemektedir. Örneğin veri yoğunluğunun ani ve çok yoğun arttığı durumlarda, veri düzlemine gelecek olan yeni paketlerin sayısı da artmaktadır. Bu durum ise veri düzleminin gelen akışlara uygun kural bulamamasına ve hangi kuralın uygulanacağına karar vermesi için kontrolör ile iletişimi sıklaştırmasına neden olmaktadır. Böylece kontrolörün çalışma hızı, veri düzlemi ile arasındaki iletişimin artış hızına yetişememekte ve tıkanıklığa sebep olmaktadır.

Kontrolör ile veri düzlemi arasında yaşanan tıkanıklık, kontrolörün gelen isteklere cevap verme süresini arttığı gibi kontrolöre gelen paketlerin düşürülme oranını da arttırmaktadır. Bunun yanında, artan cevap süresi, uçtan uca gecikmeyi (e2e latency) de arttırmaktır. Bu sorunları çözmek için, bu çalışmada kontrolöre uygulanmak üzere bir "Akış Yönetim Birimi" önerilmiştir. Önerilen bu birim kendi içerisinde Kabul ve Öncelik Yönetim Birimi olmak üzere iki alt birime sahiptir. Her iki alt birimde ise yeni nesil 5G ağlardaki farklı tiplerdeki akışların (URLLC, eMBB, mMTC) ihtiyaçları düşünülerek herbir akış için kuyruk önerilmiştir. Kabul Yönetim Birimi'nde Loss Ratio-Based Random Early Detection (LRED) algoritması çoklu kuyruk yapısına ve bir sonraki birim ile iletişime geçecek şekilde değiştirilip kullanılmıştır. Öncelik Yönetim Birimi'nde ise öncelik yönetimi için ağaç yapısı önerilmiştir. Önerilen bu ağaç yapısı ile hem farklı tipteki akışların farklı hız ihtiyaçlarına göre öncelik değerleri düşünülmüş hem de kuyrukların yakın gelecekteki doluluk oranları düşünülmüştür. Ancak, öncelik yönetimi için ağaç yapısının kurulması, uygun eyleme karar vermek için ağaç üzerinde gezileceğinden zaman kaybına neden olabilmektedir. Bu zaman kaybını önlemek için ağaç yapısındaki her düğüm için "Öncelik Değeri" tanımlanmış ve bu değerlere bakarak ağaç üzerinde gezme işleminin süresini kısaltmak için bir algoritma önerilmiştir.

Öncelik Yönetim Birimi'nde kurulan ağaç yapısında, her bir katman, uygulanacak işlem sonucunda kuyrukların gelecekte oluşabilicek olası durumlarını tutmaktadır. Üç farklı akış tipi için oluşturulmuş üç farklı kuyruktan hangisine öncelik verileceği işlemi kuyruk sayısından ötürü üç farklı olasılığa sahiptir. Bu da, ağaç üzerinde her bir düğümün üç farklı çocuk düğümüne sahip olmasına neden olmaktadır. Her bir düğümün sahip olduğu öncelik değeri, hem akış tiplerinin öncelik değerleri hem de düğümün içerisinde kuyrukların boyları ele alınarak hesaplanmaktadır. Fakat ağaç üzerinde her bir düğüm için bu değeri hesaplama ve gezme işlemlerinin zaman almaması için geliştirilen algoritma, bir sonraki katman için hangi düğümün çocuk düğümleri için öncelik değeri hesabı yapılacağına karar vermekte ve bu yolla ağacı gezmektedir. Böylece uygun kararı vermek için hem tüm düğümler için öncelik değeri hesaplanmamış hem de tüm ağaç gezilmemiş olmaktadır. Bir sonraki katmandaki çocuk düğümlerinin öncelik değerlerini hesaplamak için ise, o anki katmanda hangi düğümün öncelik değeri en büyük ise o düğüm seçilmektedir. Bu da bize hangi kuyruktan paket alınıp işleme sokulacağını göstermiş olmaktadır.

Kontrolöre uygulanan "Akış Yönetim Birimi" sayesinde, yapılan performans deneyleri ile kontrolör cevap süresinde %53'e kadar iyileşme, uçtan uca gecikmede %58'e kadar iyileşme ve paket düşme oranında ise %36'ya kadar bir iyileşme görülmüştür.

Yazılım Tanımlı Ağ teknolojisi, yukarıda bahsedilen ve kontrol düzleminde yer alan yoğun ve ani trafik artışında oluşan sorundan başka veri düzleminde de soruna sahiptir. Paket sınıflandırma işlemi, klasik ağlarda da paketlere hangi işlemlerin uygulanacağına karar veren yönlendirici ve anahtarlarda uygulanmaktadır. Yazılım Tanımlı Ağ teknolojisinde her ne kadar paketlere hangi işlemlerin uygulanacağına karar verme işlemi kontrol düzlemindeki kontrolöre verilse de, veri düzlemindeki bu cihazlar hala paket sınıflandırma işlemini yapmaktadır. Çünkü, karar veremeselerde, belleklerinde kontrolörün gönderdiği kuralları tutmaktadırlar ve gelen paketlere hangi işlemin uygulanacağına karar vermek için bu kurallara bakmaktadırlar. Böylece paket sınıflandırma işlemi, yazılım tanımlı ağlar için de geçerliliğini korumaktadır. Fakat, yazılım tanımlı ağ teknolojisinin farklı trafik tiplerine hizmet veren servislerin ağ üzerinde yer alıp kolay işlem yapmasına olanak sağlaması, bu kuralların hem sayısının hem de paketlerle karşılaştırma işlemi sırasında bakılan alan sayısının artmasına neden olmaktadır. Artan bu karmaşıklık ise paket sınıflandırma işleminin, yazılım tanımlı ağlar için önemini arttırmakta ve veri düzleminde soruna neden olmaktadır.

Paket sınıflandırma işlemi, klasik ağlardan beri var olan bir işlem olmasından dolayı, literatürde paket sınıflandırmanın hızlı yapılabilmesi çokça çalışma bulunmaktadır. Bu çalışmalar donanım bazlı ve yazılım bazlı olmak üzere iki ana gruba ayrılabilir. Donanım bazlı çalışmalarda Ternary Content Addressable Memory (TCAM) en çok kabul gören teknolojidir. Bu teknolojinin yanısıra CAM temelli olarak Binary CAM (BCAM), Field Programmable Gate Array (FPGA) ya da Graphics Processing Unit (GPU) teknolojileri kullanılarak yeni yöntemler geliştirilmektedir. Ancak donanım bazlı çözümlerin artan kural sayısı ve kural alan sayısı ile birlikte ölçekledirilebilir olmaması ve doğası gereği belirli bir donanımı gerektirmesi bu çözümleri her durumda uygulanabilir olmaktan çıkarmaktadır. Ayrıca en çok kabul gören teknoloji olan TCAM'in çok fazla enerji tüketmesi, enerjinin verimli kullanılması gereken durumlar için dezavantaj olarak görünmektedir.

Donanım bazlı çalışmaların aksine, yazılım bazlı çalışmalar altta çalışan donanımdan bağımsız olarak uygulanabilmektedir. Yazılım tanımlı çalışmalar ise kendi içlerinde

alan uzayı bazlı (tuple space based) ve karar ağacı bazlı (decision tree based) olmak üzere ikiye ayrılabilir. Karar ağacı bazlı çalışmalar, ağaç yapısını oluşturabilmek için kuralların alanlarının oluşturduğu uzayı bölmektedir. Uzayı bölme yöntemlerine göre kesme (cut) ve ayırma (split) olmak üzere ikiye ayrılabilir. Ancak, bu yöntemleri kullanan çalışmalar karar ağacını oluştururken; her bir alanda farklı kurallar olacak şekilde bölemedikleri için kural tekrarı sorunu oluşmaktadır. Bu sorunda, herhangi bir kural, ağacın birden fazla düğümünde yer alabilmektedir. Bu durum ise hızlı bir paket sınıflandırması sunan karar ağaçlarının kural güncelleme süresinde aynı başarıyı gösterememesine neden olmaktadır. Kural tekrarı sorununu ortadan kaldırmak için geliştirilen kural bölütleme çalışmaları ise çalışma süresi olarak kural alan sayısına bağlı oldukları için karar ağaçları kurulma süresini yavaşlatmakta, dolayısıyla da sürekli değişen trafiğe sahip olan ağlarda paket sınıflandırma işlemi ile kural güncellemesinin yavaşlamasına neden olmaktadır. Kural bölütleme sırasında alan sayısından bağımsız olan çalışmalar ise bazı kural alanlarının tamamını veya bir kısmını yok saydıklarından yanlış bir bölütlemeye, dolayısıyla da yanlış bir sınıflandırmaya neden olmaktadır.

Alan uzayı bazlı çalışmalardan en eskisi ve OpenFlow vSwitch içerisinde uygulananı ise Tuple Space Search (TSS) yöntemidir. TSS'nin OpenFlow tarafından kabul görmesinin en büyük nedeni sağlamış olduğu hızlı kural güncellemesidir. Çünkü karar mekanizmasının kontrolörde olduğu yazılım tanımlı ağ teknolojisinde, anahtarlarda yer alan kuralların hızlı bir şekilde güncellenmesi büyük önem taşımaktadır. Ancak sağlamış olduğu hızlı güncellemenin aksine, TSS gerektiğinde kuralların tüm alanlarını karşılaştırmak durumunda kalmasından dolayı yeterince hızlı paket sınıflandırması yapamamaktadır. TSS dışındaki alan uzayı bazlı çalışmalar ise ya birden fazla hash tablosu oluşturma ya da ayrı tablolarda olması gereken kuralları aynı tablolalara koyma dezavantajlarına sahiptirler.

Bu çalışmada ayrıca, hızlı bir paket sınıflandırması ve kural güncellemesi için karar ağacı temelli bir yöntem sunulmuştur. Ancak yukarıda bahsedildiği üzere hızlı bir paket sınıflandırmasının yanında hızlı bir kural güncellemesinin gerçekleştirilmesi için karar ağacı yöntemlerindeki kural tekrarı sorununun çözülmesi gerekmektedir. Fakat çözülmesi için önerilecek kural bölütleme yönteminin ise kural tekrarı sorununa yol açmayacak olmasının yanında çalışma süresini hızlandırması için kural alan sayısından bağımsız olması gerekmektedir. Bu bağımsızlığı sağlarken ise yanlış bir paket sınıflandırmasına yol açmaması için bölütleme sırasında kural alan sayısından bağımsız olmasına rağmen kural alanlarının karakteristiklerini yansıtması gerekmektedir. Bu bağlamda, bu tezde sunulan kural bölütleme yöntemi kural tekrarı sorununu ortadan kaldırırmakta ve çalışma süresi olarak kural alan sayısından bağımsızlık sağlarken kural alanlarının tümünün karakteristik özelliklerini göz önüne almaktadır.

Karar ağaçları oluşturmak için kullanılan kural bölütleme yöntemleri, kuralları birbirlerinden ayırırken kuralların Kartezyen Koordinat düzleminde kaplamış oldukları alan karşılıklarını kullanmaktadır. Bu bağlamda, kural alanları, koordinat düzleminin eksenlerini temsil ederken kuralların bu uzayda kapladıkları alanlar ise aslında eşlebilecek olası tüm paketleri temsil etmektedir. Çünkü kurallar koordinat düzlemi gösteriminde bir alanı temsil etmekte iken gelen paketler ise bu uzaydaki noktaları temsil etmektedir. Bu durumda, paket sınıflandırması, paketlerin, kuralların koordinat düzlemindeki alanlarının içerisinde olup olmadığının cevabı olmaktadır. Kural tekrarı sorunu ise koordinat düzleminde alanları kesişen kuralların birbirlerinden alanları kesişmeyecek şekilde ayrılamamasından kaynaklanmaktadır.

Tezde sunulan kural bölütme yöntemi de kuralların ve paket sınıflandırmasının yukarıda anlatılmış olan koordinat düzlemindeki karşılığından faydalanmaktadır. Paket sınıflandırmasının koordinat düzlemindeki karşılığının anlatımından da görüleceği üzere paketin herhangi bir kuralla eşleşmesi için paketin koordinat düzlemindeki her bir eksendeki değerinin, kuralların ilgili eksene karşılık gelen her bir bölgesinin koordinat düzlemindeki başlangıç ve bitiş değerlerinin arasında olması gerekmektedir. Tezde sunulan kural bölütme yönteminde ise bu özellikten faydalanılarak bahsi geçen eşitsizlikler her bir kural alanı veya eksen için alt alta yazıldıktan sonra toplanmıştır. Elde edilen yeni ve tek eşitsizlik ise bize paket sınıflandırmasının ve kuralların tüm alanlarını göz önüne alarak tek bir düzlemdeki karşılığını vermektedir. Tek bir düzlemde elde edilen bu karşılık ise kural bölütleme sorununu, kolayca alan bölütleme (interval partitioning) sorununa dönüşmüştür. Bu nedenle, kural bölütleme için alan bölütleme yönteminde kullanılan klasik Greedy algoritması sunulmuştur. Burada amaç ise kuralların tek boyuttaki karşılıklarından yararlanarak en az sayıda bölüt oluşacak şekilde kuralları birbirlerinden ayırmaktadır. Elden edilen kural bölütleri içerisindeki kurallar ise, alan bölütleme işleminin doğası gereği birbirleri ile kesişmemekte ve dolayısıyla da her bir bölüt için oluşturulan karar ağaçlarında kural tekrarı sorunu ortadan kaldırılmış olmaktadır. Bunun yanında ise, kuralların tek boyuttaki bu karşılıkları kullanırak yapılan bölütleme işlemi, tüm kural alanlarının karakteristik özelliklerini yansıtırken, çalışma süresi olarak da kural alan sayısından bağımsız halde getirilmiş olmaktadır.

Kuralların ve paket sınıflandırma işleminin çok boyutlu koordinat düzlemindeki karşılığının tek boyuta indirilmesi ve bu tek boyut üzerinden kural bölütleme işleminin alan bölütleme işlemine dönüştürülmesi sayesinde elde edilen kazanımlar, bize karar ağaçların daha hızlı kurulması, paket sınıflandırmasının daha hızlı yapılması ve kural tekrar tekrarı sorunun ortadan kalkması nedeniyle de daha hızlı kural güncellemesinin yapılması olanağını sunmaktadır. Yapılan simülasyon neticeleri ile de bu kazanımlar doğrulanmış ve literatürde yer alan en hızlı iki yöntemden daha iyi sonuç edildiği görülmüştür. Yapılan simülasyon sonucunda, kural bölütleme süresi ve kural güncelleme süresi olarak PartitionSort yönteminden sırasıyla %88 ve yüzde %15'e kadar iyileşme elde edildiği; paket sınıflandırma süresi olarak da %50'ye kadar iyileşme elde edildiği görülmüştür.

1. INTRODUCTION

1.1 Software-Defined Networking

Software-Define Networking (SDN) is a network management approach that has emerged to dynamize the network management and make it programmable[1]. However, this approach changes the existing traditional network architecture. Processes to be made to the incoming packets or routes of them are decided by the network elements (routers or switches) which these packets reach in traditional network architecture. SDN takes this decision authority from these network elements and creates two different planes: the data plane and the control plane. It gives the decision authority to the controller device in the control plane. The controller is in a central position to dominate the entire data plane. While communicating with the data plane, the controller sends information for controlling the data plane and the decisions about the packets which are coming to the data plane. Also, it receives information such as rule requests when the new packets come to the network, traffic information, the topology of the network. On the data plane, there are switches and routers with no decision authorities. These elements apply the instructions which are coming from the control plane.

There were attempts to make the networks more programmable in the past. Because of these attempts, the historical road for SDN starts from the early-to mids 1990s. And we can separate this road three parts: (1) the emergence of programmable functions (to the 2000s); (2) decoupling the control plane from the data plane (to 2007); (3) development of OpenFlow API and network operation system (to present) [2]. Among these improvements, OpenFlow API [3] is a milestone for SDN. It plays a critical role in SDN, even if it was first created for campus networks [3]. Because it enables the communication between the control plane and data plane. After its first appearance in 2008, the first specification came up on December 31, 2009 thanks to Internet organization *openflow.org*. After this specification, other specifications have released by the Open Networking Foundation (ONF) [4]. The last specification for OpenFlow

is the OpenFlow Switch Specification Version 1.5.1 [5]. The birth of OpenFlow brought along with it the acceleration in developing network operating systems for SDN. Among these operating systems, NOX (the first one) [6], POX (python version of NOX) [7], the Beacon [8], Floodlight [9], RYU [10], OpenDaylight [11] and Open Network Operating System (ONOS) [12] are the most accepted and used ones.

SDN has its own problems like other network architectures. These problems can be divided into two groups: problems in the control plane and problems in the data plane. The most important problem in the control plane is the centrality of the SDN controller. Because this causes scalability and resiliency issues. To solve this problem, multiple controllers can be implemented in the control plane. But, this is also another research area because there are specific problems about that how many controllers are enough and how they are distributed and synchronized among themselves [20]. Also, because of the centrality and openness criteria of SDN, security problems about the control plane has been getting attention [21]. Lastly, this centrality problem is the reason for congestion between the control plane and data plane, which eventually brings an increase in latency and drop rate of packets in the data plane. Apart from control plane problems, the data plane has its own problems too. The most important one is the capacity of OpenFlow switches because there may be lots of users or devices in the network thanks to the advantages of SDN. The capacity of OpenFlow switches isn't scalable with this increase in users or devices to hold enough rules. This problem creates its own research area in the data plane, such as rules placement [22]. Also, e2e latency in the data plane is a problem because of long lookup duration to find proper action(s) for an incoming packet in OpenFlow switches [23]. This thesis focuses on specific problems in the control plane and data plane.

1.2 Congestion Problem of SDN Controller Under Heavy Traffic

Applications or network services can use customized resources, thanks to SDN. This leads to the usage of SDN in 5G to meet the requirements of different flow types in 5G. Apart from the expectation of 1~20 Gbps throughput and less than 1 ms latency from 5G [24], heterogeneous flow types need different throughputs and latencies in 5G. For example, Enhanced Mobile Broad (eMBB), Massive Machine-Type Communication (mMTC), and Ultra-Reliable and Low-Latency Communications (URLLC) require 4 ms, 10 ms, and 0.5 ms latency, respectively [25]. However, these

requirements cannot be met when there are ultra-high demands in the SDN network because the communication channel between the control plane and data plane congests in ultra-dense scenarios.

In ultra-dense scenarios, newly incoming flows increase in the data plane. This causes an increase in PACKET_IN messages that are sent from switches to the SDN controller to ask an action for newly incoming flows. As a result, the response time of the SDN controller increases and causes congestion between the data plane and the control plane. This congestion also causes an increase in e2e latency and the number of ignored PACKET_IN messages. We create a network described in Section 2.2 and don't implement any solution to examine the increase in the response time of the SDN controller. As seen in Figure 1.1, when the number of hosts increases, that means an increase in newly incoming flows, the response time of the SDN controller also increases and surpasses a target value after a specific amount for the number of hosts. The first aim of this thesis is to decrease the response time of the SDN controller for different flow types of 5G to the target area in ultra-dense scenarios. We determine 10 ms for a target value because the highest latency requirement is 10 ms for mMTC flow type.

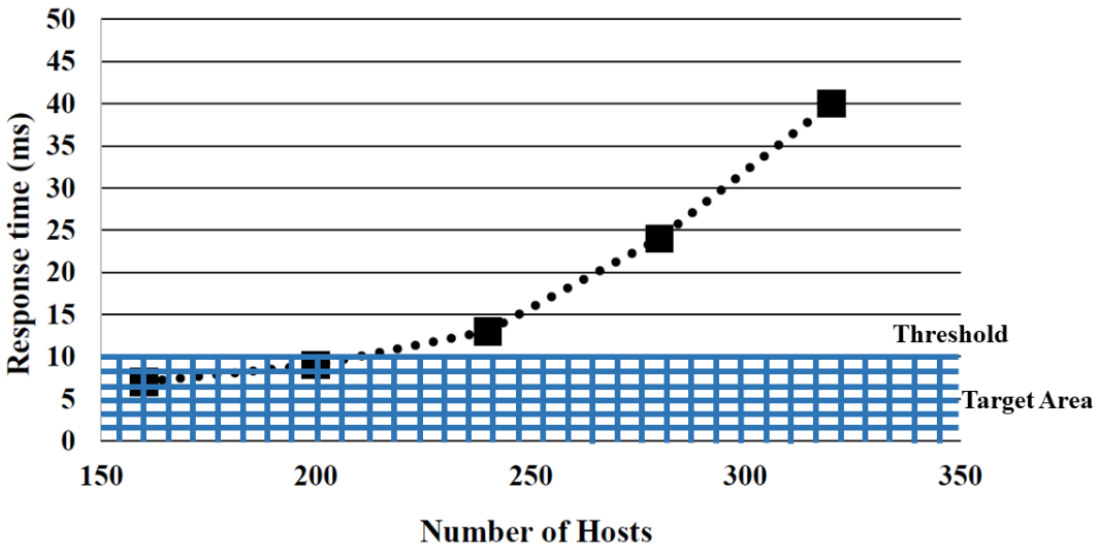


Figure 1.1: Controller Response Time vs Number of Hosts.

1.3 Packet Classification Problem in Data Plane

All packets in a network are not the same with each other; as a result, each different packet requires the most proper action to be done for it. Packet classification is an

essential function to find the appropriate action in all networking paradigm. Also, more diverse services can find a place for themselves in a network as new networking technologies emerge. But more diverse services require more diverse actions. As a new paradigm, SDN with the OpenFlow standard tries to meet this requirement by increasing the number of supported fields up to 45 [26]. But, this improvement increases the importance of packet classification for OpenFlow even more, because it is more difficult to find a proper action/rule for a packet in this diversity. In addition to this difficulty of classification in OpenFlow, incoming flows need fast processing in an OpenFlow switch because of constrains of communication services in real-time [27]. For example, URLLC service in 5G needs 0.5 ms latency at most [25]. Also, the SDN controller can easily update a rule in a switch thanks to its centrality and softwarization feature. But, rule updating also needs to be done quickly because of the time constraints mentioned above.

Decision tree methods are the most preferred methods in the literature when we want a fast packet classification. But, the rule updating time of these methods is very high because most of these methods separate rules by cutting or splitting search spaces of rules. As a result, some rules have one or more replica in leaf nodes of the decision tree, known as ’rule replication problem. Rule partitioning methods combined with decision tree methods, minimize the rule replication problem, or eliminate it. Thus, they have fast classification and rule updating time. However, while partitioning rules, they depend on the number of rule fields or ignore most rule fields for fast partitioning. On the other hand, Open vSwitch [28] prefers to use the Tuple Space Search (TSS) [29], which is the most popular tuple-space-based solution. TSS has a very fast rule updating time, but it classifies packets slower than decision tree-based methods. Because there are lots of hash tables as a result of ineffective rule partitioning [27]. To solve these problems, we first aim to solve the rule partitioning problem in an effective and fast way. Then, we use the HyperCuts method [30] to construct a decision tree for each ruleset created by the proposed rule partitioning method for fast classification and updating.

1.4 Contributions

1.4.1 Flow management engine for the sdn controller under ultra-dense demands

We first try to decrease the response time of the SDN controller in the thesis. By decreasing response time, we aim to reduce e2e latency and the number of the ignored PACKET_IN messages that will result a decrease in drop packet rate. To do these, we propose a Management Engine, which has two modules in: Admission and Prioritization modules. In the admission module, we use and change the Loss Ration-Based RED (LRED) [31] method accordingly our need for multi-queue status. In the prioritization module, we implement a tree structure that holds possible next states of queues in each level and holds a priority value in each node, which is calculated using priorities of each flow type and each queue lengths. To give priority to a queue/flow type, we travel through the tree by looking at priority values of each node. So our contributions include:

- To avoid congestion between the control plane and data plane, we implement and change the LRED algorithm in the admission module in a way that it can consider multiple queues in itself at the same time and states of queues in a different module.
- We propose a novel priority value in the prioritization module. Thanks to this priority value, we can consider each delay requirement of flow types of 5G and queue fullness of each flow in this module. As a result, we fairly give priority to each flow type and achieve a decrease in e2e latency.

1.4.2 Interval Partitioning for Packet Classification in OpenFlow vSwitch

In this part of the thesis, we primarily focus on the rule partitioning, because it is a key factor for fast packet classification and rule updating in both decision tree-based and tuple-spacebased solutions. To solve the rule partitioning, we convert this problem to an interval partitioning problem and propose a classic Greedy Algorithm. Thus, we eliminate the rule replication problem. After partitioning, we construct decision trees using the HyperCuts method [30] and order these trees according to the highest priority values they have for fast classification. So, our contributions include:

- Running time of the proposed rule partitioning method is not dependent on the number of rule fields while we consider the characteristics of each rule field. As a result, we decrease construction time of decision trees.
- We eliminate the rule replication problem in each decision tree by converting the rule partitioning problem to the interval partitioning problem. Consequently, we decrease the rule updating time.
- We construct wider and shorter decision trees thanks to both the HyperCuts method and our proposed partitioning method. Thus, we decrease the packet classification time.

We organize the rest of the thesis as follows: We seperately give a literature surver about the congestion problem and packet classification problem in Section 2. In Section 3, we explain the proposed management engine for the congestion problem in detail with the simulation results. Also, In Section 4, we explain the proposed rule partitioning method for fast packet classification and rule update in OpenFlow vSwitch with the performance evaulation of our method. We conclude the thesis in Section 5.

2. LITERATURE SURVEY

2.1 Current Solutions for Congestion Problem in SDN

Response time of the controller, e2e latency, and packet drop rate increase because of ultra-high demand on the network and the controller's centrality feature. To solve this problem, using multiple controllers in a way that they are distributed throughout the network is the most implemented solution in the literature [32]. Authors in [32] try to equilibrate loads of controllers among them using the response time of each as a threshold in the distributed controller architecture. Authors in [33] aim to decrease the response time of the controller by trying to reduce the load of it. They send and load rules that contain wild-card bits to the switch(es) for some of the flows beforehand will be these flows in the switch(es). In [34], a rounding-based algorithm is used to balance the demands in the links and controller. [35] dynamically assigns controllers to the switches and finds a Nash stable point after solving the stable matching problem for these assignments. [36] considers reliability and response time together and define a new metric Quality of Controller (QoC). After that, authors map switches and controllers with each other using QoC. [37] makes one controller responsible for many switches by considering the processing capacity of each controller. Authors in [38] change the assignment of a switch before congestion between this switch and controller by predicting the future amount of flows that will come to this switch. [39] implements a layer named “flowcache” between the control plane and data plane to cache some requests coming from switches to decrease the load of the controllers. Finally, authors in [40] implement different controllers in architecture in a way each controller is responsible for different works. But, these works aren’t adaptable for 5G to implement to meet the requirements of different flow types and prioritize these flow because they don’t take into account these requirements. Also, solutions of these works don’t consume as little time as latency requirements of 5G flow due to do their processes.

2.2 Current Solutions for Packet Classification Problem in SDN

Methods for packet classification can be categorized into two main groups: hardware-based and algorithmic-based. Hardware-based solutions usually choose hardware derived from Content Addressable Memory (CAM), such as ternary CAM (TCAM) [41], [42], binary CAM (BCAM) [43]. Also, some hardware-based solutions use Field Programmable Gate Array (FPGA) [44] or Graphics Processing Unit [45] as a hardware technology. Hardware-based solutions are faster than algorithmic-based solutions, especially TCAM-based solutions. However, TCAM-based solutions consume lots of power and aren't scalable with an increase in the number of rules. Besides, we cannot always use hardware-based solutions for every situation because of the requirement for specific hardware.

Algorithmic-based solutions are mainly divided into two as decision tree-based and tuple-space based. Decision treebased methods have three different ways for separating search spaces of rules: cutting (HyperCuts [30], EffiCut [46]), splitting (HyperSplit [47], SmartSplit [48]) and hybrid usage of both (CutSplit [49]). EffiCut minimizes the rule replication problem by constructing multiple decision trees compared to HyperCuts; however, it is slow in classification due to the large number of trees. SmartSplit divides the rules using the Efficut method and creates a decision tree using either HyperSplit or HyperCuts methods according to the properties of the resulting rulesets. As a result, it classifies packets faster than other methods; but it does not allow a fast rule update due to its feature of complex decision tree construction. CutSplit divides the rules based on the properties of some rule fields. As a result, it can cause mismatches in packet-rule matching. In addition to these decision tree methods, PartitionSort [50] defines a sortability function and partitions the rules using this function. Thus, it has a balance between the rule update and packet classification time. However, the running time of rule partitioning depends on the number of rule fields. This makes this method not a scalable solution with the increasing number of rule fields. TSS [51] method, which is the most common of Tuple-Space based methods, offers a fast rule update. However, the packet classification is slow due to lots of field comparisons when necessary and high number of hash tables. The TupleMerge [27] method that improves TSS has faster packet classification and rule update than PartitionSort as it combines some hash tables. However, it combines rules that must be separated by ignoring some bits in the rule fields.

As seen, decision tree methods that do not construct multiple trees cause the rule replication problem. When they construct multiple trees, they are not scalable because they depend on the number of rule fields during the rule partitioning, or they cannot eliminate the rule replication problem. When they partition the rules independent from the number of rule fields, they can cause errors in matching because they ignore some rule fields and don't reflect the characteristics of ignored fields in rulesets after partitioning. Likewise, tuple-space based methods may result in a large number of rule groups as a result of rule partitioning or cause inaccurate rulesets if they sacrifice from characteristics of rule fields.

3. FLOW MANAGEMENT ENGINE FOR THE SDN CONTROLLER UNDER ULTRA-DENSE DEMANDS

3.1 Proposed Flow Management Engine

The recommended management engine can be seen in Figure 3.1. The engine consist of two separated but connected steps: admission and prioritization steps. In the admission step, we implement three different queues for three flow types of 5G: eMBB, mMTC, and URLLC flow. In this step, we changed the LRED algorithm in a way that we take into account both queues in this step and queues in the prioritization step. If there is a signal (s) that is coming from the prioritization step and shows one of the queues in this step is full, we drop a message from the queue in the admission step that corresponds to the queue created for the same flow type in prioritization step. The admission step would send the request (t) to the prioritization step if that request didn't drop. In the prioritization step, we give priority to flows considering both delay requirements and queue fullness of these flow. We will explain the admission step and prioritization step in detail in Section 3.1 and Section 3.2, respectively.

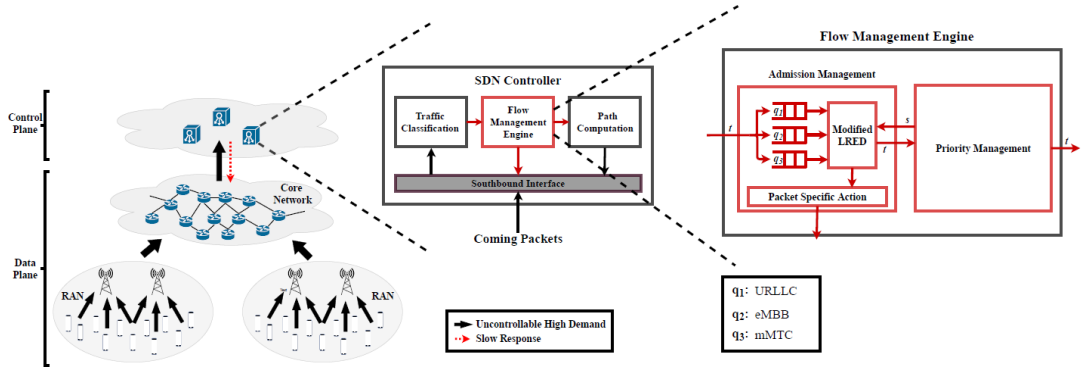


Figure 3.1: The Proposed Management Engine.

3.1.1 Admission step

We use the LRED algorithm to admit messages coming from the data plane in this step. The LRED algorithm is an active queue management method. We use this algorithm because it is fast and depends on the loss-ratio of the coming packets. How

However, we change this algorithm to take into account of our multiple queues in this step and queues in the next step given as Algorithm 3.1.

Algorithm 3.1: Admission Management Algorithm.

```
Initialize γ
Equalize s to FALSE
for one flow type j
    determine drop probability r_j with equation (3.3)
    create v between 0 and 1 in a uniformly randomize manner
    if queue is full
        send the packet_in message to the switch
        add one to m_nj
    else
        if s_j is TRUE
            send the packet_in message to the switch
            add one to m_nj
        else if v < r_j
            send the packet_in message to the switch
            add one to m_nj
end
```

As seen in Algorithm 1, we look at the queue fullness of each queue in this step before looking at any other condition. If the queue is full, we send the newly incoming packet_in message for that flow type back to switch. If the queue is not full, we check queue fullness again; but this time, the algorithm looks at the queues in the prioritization step. That means the algorithm looks the value of *s*, whether it is TRUE or not. If *s* is TRUE, the algorithm again sends the incoming packet_in message back to switch. If *s* is FALSE and *v*, which is a random value produced before is less than the drop probability value r_j, which is calculated earlier, we again reject the incoming packet_in message for the related flow type. In that point, by saying drop probability, we mean sending back or rejecting the incoming packet_in message. So we use the words 'drop' and 'rejection' interchangeably from now on. After each rejection of the packet_in message, we increase the number of the rejected message by one to use this value the calculation of drop ratio. We use the equation 3.1 to calculate the drop ratio (f_j) for each flow below:

$$f_j(t) = \frac{\sum_{n=0}^{T-1} m_{nj}(t-n)}{\sum_{n=0}^{T-1} m_{aj}(t-n)} \tag{3.1}$$

In the equation 3.1, $m_{aj}(t)$ is the total number of incoming packet_in messages for a flow type, $m_{nj}(t)$ is the total number of rejected packet_in messages for the same flow type. These two values are calculated for a period *T*.

This drop ratio is used to calculate the expected drop rate (f_j^*) for each flow type. This rate is necessary to determine a drop probability r_j for each flow type and equation of this rate as follows:

$$f_j^*(t) = w_j . f_j^*(t-1) + (1 - w_j) f_j(t) \tag{3.2}$$

where w_j is the weighting factor. After that, we calculate the drop probability for each flow type using $f_j^*(t)$, instant length of each queue l_j, and steady-state queue length l_{sj} as follows:

$$r_j = f_j^*(j) + \gamma \sqrt{f_j^*(t)}(l_j - l_{sj}), \quad where\ \gamma > 0 \tag{3.3}$$

where γ is constant value. In that point, if we use the same l_{sj} value for all flow types in the calculation of the drop probability, there will be an unnecessary rejection of packet_in messages of the flow type with the highest arrival rate. We give the lowest l_{sj} value for this flow type differently from the original LRED algorithm [31] to prevent unnecessary rejections. Also, we give different l_{sj} values for other flow types in the manner because each flow type has different arrival rates.

3.1.2 Prioritization step

In this step, we construct three different queues for each flow type because each flow type has different delay requirements in 5G. However, we also have to decrease the duration of each message of a flow in this step. That means we have to take into account the length of each queue. To solve these problems, we define a new priority value that considers both delay requirements and length of queues of each flow. Also, we create a tree structure, and an algorithm to travel the tree to select which flow type's message should be considered first.

The created tree contains four levels, as seen in Figure 3.2. There are possible next states of each queue in each level. Nodes in the tree hold a priority value which is calculated as follows:

$$P = \max(\frac{N_1}{p_1}, \frac{N_2}{p_2}, \frac{N_3}{p_3}) \max(N_1 p_1, N_2 p_2, N_3 p_3), \tag{3.4}$$

where P is the priority value for a node, N is the length of each queue, and p_j is the priority value which represents delay requirements of each flow. These priority values for flow types is between 0 and 1. Also, the summation of these priority values equals

1, which can be shown as $\sum_{j=1}^{3} p_j = 1$. Thanks to the priority value *P* for a node, we consider both lengths of the queue and delay requirements of each flow. Also, as seen in Figure 3.2, each node has three child nodes representing the next possible states created from this node. These states are selecting from a message only from the first flow type, only from the second flow type, and only from the third flow type.

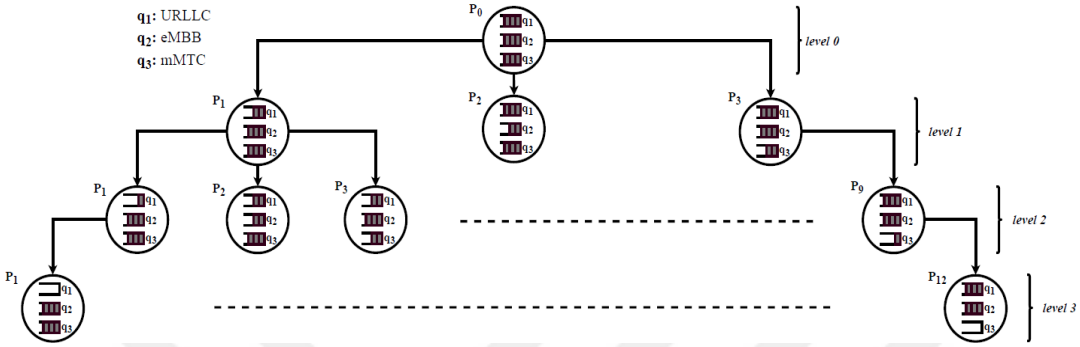


Figure 3.2: The Proposed Tree Structure.

Calculation of priority values of each node and traveling all the nodes in the tree causes an increase in the response time of the controller. To prevent this increase, we create four levels and propose an algorithm, as seen in the Algorithm 3.2. In the algorithm, we first check whether there is more than one node in which there is queue(s) whose length(es) is greater than or equal to the determined threshold value in or not. If there is more than one node, we sum the priority values of flow types whose queues exceed the threshold value. After that, we select the node whose summation of priority values is greater than other nodes' as a parent node. If there is one node in this situation described above, we select that node as a parent node. However, if no node is in that situation, we calculate the priority value *P* for each node at the same level using equation 3.4. After that, we select the node whose priority value *P* is greater than other's queues exceed the threshold value with higher priority values. The threshold value is determined using 9(whole queue length)/10 because of the ratio between the possible highest priority value for a flow type to has is 0.7 and possible lowest value 0.1. As a result, we expect a selection from all queues till the threshold. Another proof for considering both queue length and delay requirements is the selection of the node with the maximum priority value *P*. As seen in the equation 3.4, the max(.) function at the right pushes to select a message from the queue if the length of this queue is the maximum. On the other hand, the max(.) function at the left pushes to select a message from the queue, which holds messages for the flow type whose delay tolerance is the

minimum. Because we give the highest priority value to this flow type to this flow type.

Algorithm 3.2: Prioritizaion Algorithm.

Initialize N_1, N_2, N_3 for the starting level
for each level *s* starting from the level 1 in the tree
    determine possible lengthes *N* for each queue in all nodes
    if more than one nodes have queue(s) whose length(es) are greater than or equal to the threshold value
        select the node as a parent, which has more flow types with higher priority values
        add this node to the decision path
    if only one node has queue(s) whose length(es) are greater than or equal to the threshold value
        select the node as a parent
        add this node to the decision path
    else
        determine priority values *P* for each node using the equation (3.4)
        select the node a parent with the highest *P*
        add this node to the decision path
execute the decision path
end

3.2 Performance Results

We evaluate our proposed method using a single machine with Ubuntu 18.04 LTS, Intel Core i7, 12.00 GB RAM. We construct a topology with 20 OpenFlow switches and 2 controllers. Each controller is responsible for half of the switches and 8 hosts connect to each switch. We choose to use POX [7] as an operating system for controllers. We created three flow types with different characteristics to reflect URLLC, mMTC, eMBB services. In this context, we give different priority values for these services and create different packet sizes as seen in Table 3.1. Also, we use the "waiting time calculation" in [52] to create these flows with different distributions because eMMB is UDP, URLLC and mMTC flows are TCP flows. Other details about the performance parameters can be seen in Table 3.1.

We compare our method with two different methods: only using First In First Out (FIFO) method (method 1) in the admission step and only using our LRED method in the admission step (method 2). As seen, we don't implement a prioritization step in these two methods. Because we want to compare the effect of both admission and prioritization steps separately. We compare our management engine with these methods in controller response time, e2e latency, and drop rate for each flow type. Du

During the experiments, we add 40 hosts to network in each experiment that means 2 additional hosts for each switch.

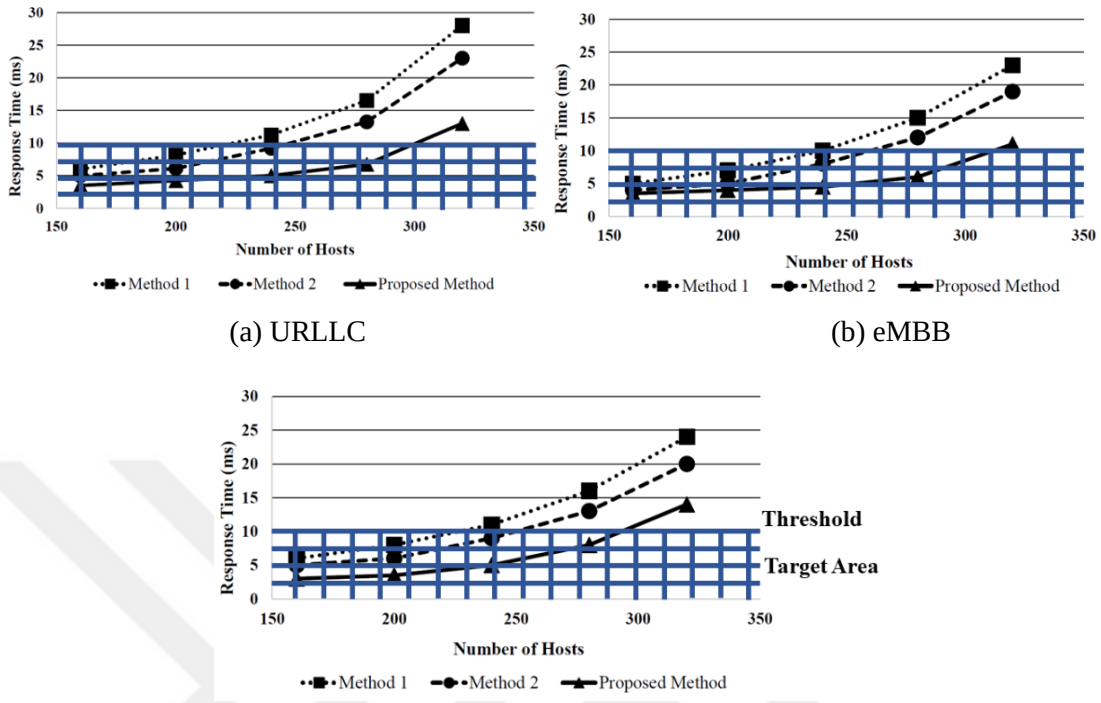
Table 3.1: Performance Parameters.

Parameter	Value
Mininet Version	2.2.2
OpenFlow Version	1.3
Hosts per Switch	[8-16]
Queue Length	1000 packets
Arrival Rate per Flow	500 packets / sec
Priorit values	0.5, 0.3, 0.2 (URLLC, mMTC, eMBB)
Packet Sizes	100, 80, 70 byte (URLLC, mMTC, eMBB)

For the average controller response time, we get the best results, as seen in Figure 3.3. Also, our method results are close to the target area, which is shown in Fig. 1.1 because firstly, in the admission step, we reject the messages if the queue for that flow type is full in the prioritization step. Another reason for getting these results is our selection method described in Algorithm 3.2 and equation 3.4. In the algorithm, we firstly select messages from the queues whose lengths exceed the threshold value. But, while selecting, we also take into account delay requirements of flow types by looking at their priority values. Secondly, we select messages by looking at the priority value of nodes calculated using equation 3.4. Thanks to max(.) functions in this equation, while we consider priorities of flow types, we take into account of the duration of messages in the prioritization. Also, differences between the results for flow types are not huge thanks to selection method in the prioritization method as seen in Figure 3.3. But, the best results among the flow types is for the URLLC service with approximately 53% better results than method 1 and 43% better results than method 2 when there 3200 hosts, as seen in the Figure 3.3 (a). Because, it has the highest priority value.

Our method gives a response to incoming packet_in messages in a fast and dynamic way thanks to our changed LRED method in the admission step. That also brings us an advantage in preventing congestion between the control plane and the data plane. As a result of this advantage, we decrease e2e latency for all flow types, as seen in Figure 3.4. We improve e2e latency for URLLC flow type with the amount of 58% from method 1, and 50% from method 2 when there are 3200 hosts; but this improvem improvement is lower for other flow types. Because we give different priority value of

improvement is lower for other flow types. Because we give different priority value of them and URLLC flow type has the highest priority in the equation 3.4.



(a) URLLC (b) eMBB

(c) mMTC

Figure 3.3: Average Response Time vs Number of Hosts.

LRED method considers the drop rate of the packet while calculating the drop probability. So, our method also considers the drop rate, which means the rejection rate of packet_in messages, as seen in equation 3.3. As a result, our method gives the best results for all flow types in terms of drop rate, as seen in Figure 3.5. However, the improvement from method 2 isn't huge, like the improvement of method 2 from method 1. For example, the drop rate of mMTC flow is less in method 2 than method 1 with the amount of 21% but is less in the proposed method than method 2 with the amount of 3%. Because, when we implement the admission step with the prioritization step, the admission step considers the queue fullness in the prioritization.

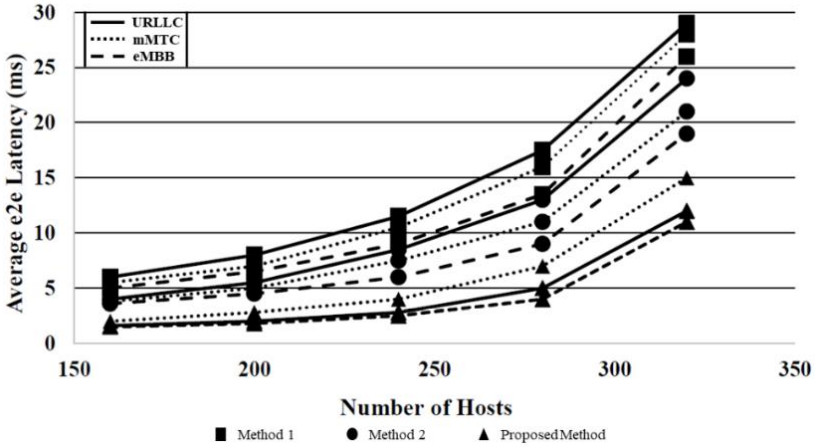


Figure 3.4: e2e latency vs Number of Hosts.

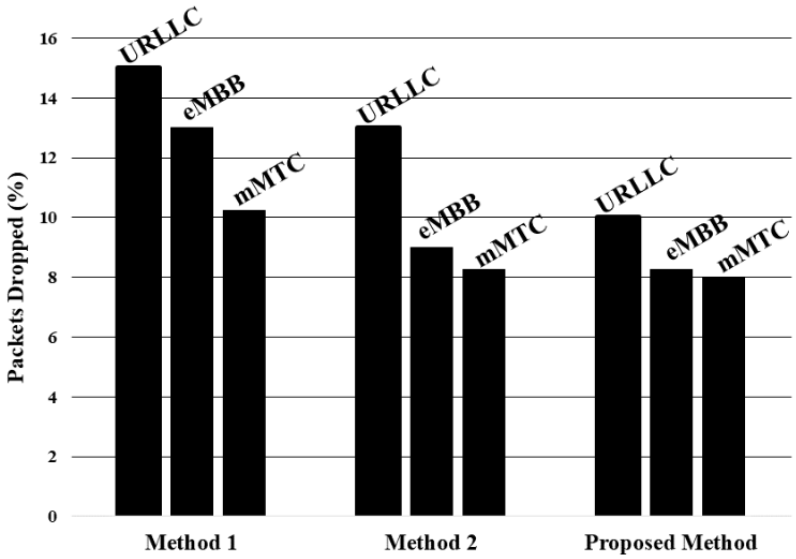


Figure 3.5: Packet Drop Rate.

4. INTERVAL PARTITIONING FOR PACKET CLASSIFICATION IN OPENFLOW vSWITCH

4.1 Proposed Method

In this section, we will explain how we develop a partitioning method considering the characteristic of rule fields but eliminating the adverse effects of the number of rule fields. After that, we describe how we can construct decision trees, classify a packet, and update a rule, respectively.

4.1.1 Proposed rule partioning method

We must overcome two main challenges in developing an effective and fast partitioning method: *i)* How can we consider characteristics of rule fields but prevent retarding effect of the number of fields in partitioning at the same time? *ii)* How can we separate rules in a way that there will be no rule replication in the decision tree constructed using one ruleset?

Before digging into these challenges, it is necessary to explain the counterpart of packet classification in geometry. It will be better to think of a two-dimensional Cartesian coordinate system to facilitate explanation. We create an example of rules in Table 4.1 to use in the explanation. When we create a Cartesian coordinate system whose axes represent rule fields, a rule forms a rectangle, and a packet forms a dot whose coordinates are values of header fields of the packet in decimal base. In that representation, when we said that a packet matches with a rule, it means that packet or dot will be in the area of rectangle shape of that rule, as seen in Figure 4.1a. The dot $P_{(x,y)}$ shows the incoming packet.

If we look specifically at the matching of the packet with the rule R5 in Figure 4.1a, we can say that coordinate values of the packet are between coordinate values of two corners of the rule represented with dots $I_{(a,b)}$ and $J_{(c,d)}$ That mathematically means:

$$a \leq x \leq c, \tag{4.1}$$

$$b \leq y \leq d, \tag{4.2}$$

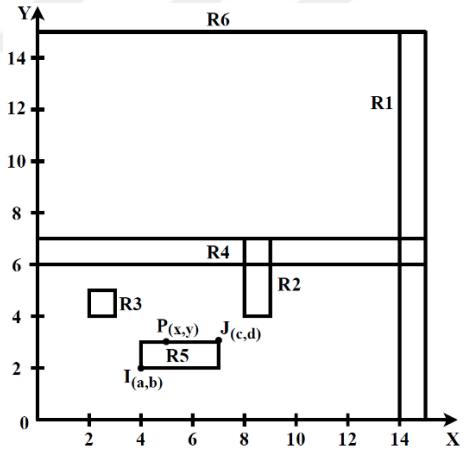
After that, we can easily sum the inequality 4.1 with the inequality 4.2 and get:

$$a + b \leq x + y \leq c + d, \tag{4.3}$$

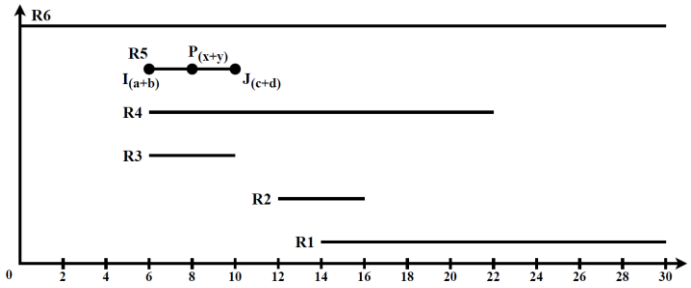
Table 4.1: Example 2D Rulelist.

Rules	Field X	Field Y	in 1D
R1	[14 15]	[0 15]	[14 30]
R2	[8 9]	[4 7]	[12 16]
R3	[2 3]	[4 5]	[6 8]
R4	[0 15]	[6 7]	[6 22]
R5	[4 7]	[2 3]	[6 10]
R6	[0 15]	[0 15]	[0 30]

The question at this point is: what is the meaning of the inequality 4.3 in a geometric perspective? When we interpret the inequality 4.3 in one dimension, we can see that if a packet matches with a rule, it means the packet must be on the line, which represents the related rule, as seen in Figure 4.1b. If we look at the whole picture of Figure 4.1b, the rule partitioning problem becomes an interval partitioning problem. At this point, instead of using as few sources as possible, we try to use as few rulesets as possible to construct decision trees for each ruleset.



(a) Rules in 2D.



(b) Rules in 1D.

Figure 4.1: Rules Before Partitioning and Packet Classification.

We can solve the first challenge mentioned at the beginning in this section by converting the rule partitioning problem to the interval partitioning problem and representing the packet classification in one dimension. As seen in Figure 4.1b, we get line representations of rules by summing coordinate values of start and end points of each rule interval. That means we consider characteristics of each rule field while we obtain line representation of each rule. Besides, we propose a classic greedy algorithm solution [53] for rule partitioning, as seen in Algorithm 4.1. If we use a greedy algorithm, the limitation of the running time of the rule partitioning problem becomes *O(nlogn)*, where n is the number of the rules. As a result, we eliminate the retarding effect of the number of fields in rule partitioning.

Algorithm 4.1: Rule Partitioning Greedy Algorithm.

get rule intervals in 1D
sort intervals by starting time in a way $s_1 \leq s_2 \leq \cdots \leq s_n$
set number of ruleset r → 0
for *l* = 1 to *n* **do**
    if rule *l* is compatible with some ruleset *k* **then**
        put rule *l* in ruleset *k*
    else
        construct a new ruleset r + 1
        put rule *l* in ruleset r + 1
        r ← r + 1
    end if
end for
end

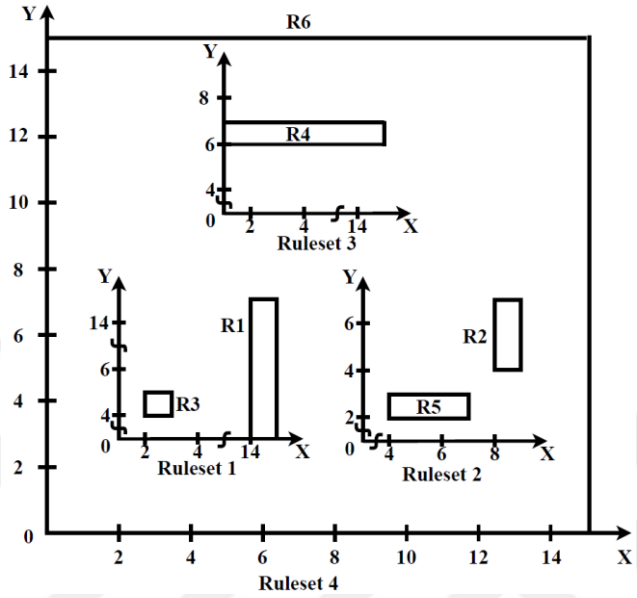
Thanks to the proposed greedy algorithm, we can solve the second challenge mentioned at the beginning of this section. We can say that the rules in one ruleset don't overlap with each other. That means the ranges for packets of these rules don't intersect with each other's ranges. Thus, there will be no rule replication in decision trees. The partitioning results can be seen in Figure 4.2a and Figure 4.2b in 2D and 1D, respectively. Before finishing this chapter, we prove the solution of the second challenge.

Lemma: Let have d-dimensional n rules R1 through RN in one ruleset obtained the proposed partitioning method. For any resulting ruleset, there will be no rule replication in the decision tree constructed by using this ruleset.

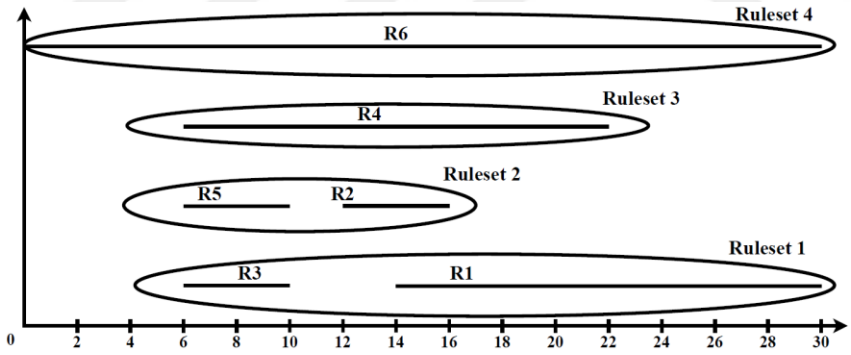
Proof: If any two or more rules in one ruleset intersect with each other's ranges, that means one incoming packet classified to this ruleset can match with these intersected rules. Let think one ruleset that is obtained the proposed partitioning method and has

two rules R1, R2. Each rule has two fields. Also, let range or interval of rule R1 in two fields be [a, c] and [b, d], respectively, and range or interval of rule R2 in two fields be [k, m] and [l, n] respectively. Lastly, let say:

$$a + b < c + d < k + l < m + n, \tag{4.4}$$



(a) Rules in 2D.



(b) Rules in 1D.

Figure 4.2: Rules After Partitioning.

We can construct this inequality because if we solved an interval partitioning problem, we know that each interval in one set doesn't intersect with each other. Let an incoming packet P has two header fields with the value of (e, f). If this incoming packet matches with both R1 and R2, that must be $a + b \leq e + f \leq c + d$ and $k + l \leq e + f \leq m + n$ But we know that $c + d < k + l$. That means $e + f$ don't be $\leq c + d$ and $\geq k + l$ at the same same time. As a result, an incoming packet can't match with these two rules at the same time. Also, that means ranges of R1 and R2 can't intersect with each

other. This proof doesn't change if we use more rules or more fields. This proof shows that we can construct a decision tree for one ruleset in a way that each leaf node of this decision tree represents different rule(s). As a result, we eliminate the rule replication problem thanks to the proposed partitioning method.

4.1.2 Classification, rule update

4.1.2.1 Classification

We must first construct a decision tree for each ruleset, which is obtained using the proposed interval partitioning algorithm before classification a packet. We will use the HyperCuts method [30] to construct a decision tree. The reason for choosing this method is a facility for constructing a wider decision tree. Thanks to this tree, we can do classification faster. Also, we can construct these decision trees in a way that each leaf node in a tree holds different rule(s) because rules in a ruleset don't intersect with each other.

We determine a condition to construct a decision tree. This condition is that number of rules in one ruleset must be higher or equal than 2 because there is a possibility of getting rulesets that have rules less than 2. After constructing decision trees, we combine the rulesets that don't allow us to construct a decision tree and make one ruleset from these separate rulesets. Then, we order the rules in the combined ruleset according to the priority order in a table holding. Also, we hold the information about the interval of each rule in 1D (as seen in Figure 4.1a) in this table.

When a packet comes, we have to determine which ruleset or decision tree we start to search for matching in. After constructing part, we order decision trees and the remaining combined ruleset according to the maximum priority value of a rule that is in each tree or ruleset like in PartitionSort [50]. Consequently, we start to search for a matching according to this order. If the packet matches and priority value of matched rule is higher than the next ruleset, we stop searching. When we search a matching in the remaining combined ruleset, we also start from the rule which has the highest priority value by looking at the 1D interval information of each rule. We stop searching after we find a match in the table.

4.1.2.2 Rule update

We use a similar approach in rule insertion and eviction operation, shortly named rule update, with PartitionSort. In rule insertion operation, we look at the decision trees in order. If a new rule can be held in one leaf node of one decision tree, we add the new rule to this decision tree. If not, we add this new rule to the remaining combined ruleset. After that, we run the interval partitioning algorithm only for the new version of the remaining ruleset. If we can construct a new decision tree(s), we construct it (them) and again combine remaining rulesets. Then, we reorder decision trees, and the remaining combined ruleset again.

In rule eviction operation, after we found which ruleset the rule is in, we evict this rule from that ruleset. If we evict a rule from a decision tree and the number of rules in this tree becomes less than 2, we add the remaining rule the combined ruleset. After these operations, we check the priority order of rulesets, and if necessary, we reorder the rulesets.

4.2 Performance Evaluation

We evaluate our method by creating synthetic rulelists using ClassBench [54], because we don't have an access to the real rulelists. We use 12 different parameter files for creation synthetic rulelists. These files include three different categories: 5 access control lists (ACL), 5 firewalls (FW) and 2 IP chains (IPC). We generate 4 different sizes of rulelists from 1k to 64k for each category.We create 5 different rulelists per each size using each parameter files. As a result, we have 240 different rulelist in total. Finally, we generate different rulelists with different number of rule fields (5, 10, 15, and 20 rule fields) for 64k rule size.

We compare our method with PartitionSort [50] and TupleMerge [27] in two scenarios: online scenarios and offline scenarios. For online scenarios, we measure packet classification and rule update times. For offline scenarios, we measure construction and packet classification times. In our experiment, classification time is the time that is needed to classify 1.000.000 packets. We generate these packets using the Trace Generator tool of the ClassBench method. Rule update time is the time that is needed to insert or delete one rule. Finally, construction time is the time that is

needed to construct the data structure for rules. In comparison, we use the average value of each performance metric for each category and size.

To evaluate our method, we use Intel i7-8750H CPU, 2.2GHz with 6 Cores, 16GB RAM running Windows 10 as an environment.

4.2.1 Online scenario

In the online scenario, we randomly select half of a rulelist to construct a data structure. After that, we insert the other halves in random order. While inserting, we randomly delete some rules from the data structure. The total number of insertions and deletions are 500.000 per each. While dynamically changing the rules in the data structure, we measure and compare packet classification and rule update time of PartitionSort, TupleMerge, and the proposed methods.

In Figure 4.3, we compare packet classification times of methods in each size while increasing the size of rulelists. As seen, our method has the best classification time for each category and in each size because our decision trees for each ruleset is wider than decision trees in PartitionSort thanks to using HyperCuts for construction even if we have more partitions. Our method is also better than the TupleMerge method because we have fewer rules in each ruleset, and we don't have to search for each rule field sometimes. We have the best improvement (up to 40% better than the TupleMerge) for the Ipc category because there are rules whose search space of fields is narrower in this category. As a result, we have a fewer number of rulesets for this category.

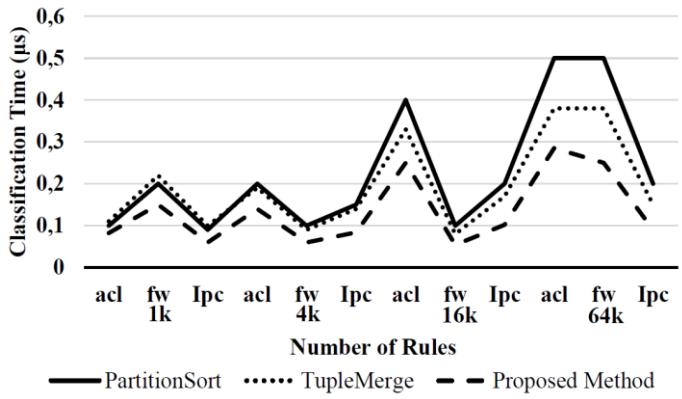


Figure 4.3: Online Classification vs Rule Size.

In Figure 4.4, we compare the rule update time of each method while increasing the size of the rulelists. As seen, our method is better than the PartitionSort (up to 15 % better), because of wider decision trees. Unlike the classification time, our method is

worse than the TupleMerge in rule update time because the TupleMerge uses hash tables like the TSS method as a data structure; but, our method isn't too worse.

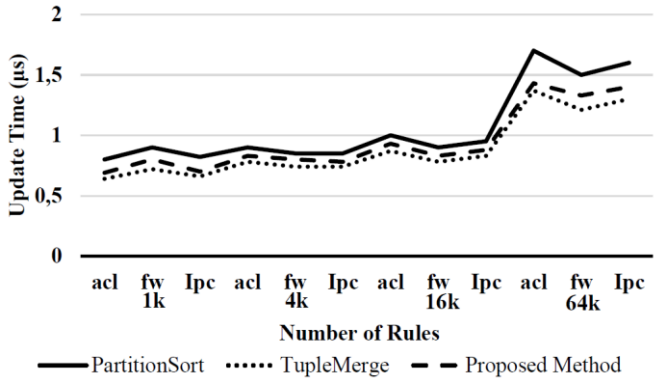


Figure 4.4: Online Update Time vs Rule Size.

4.2.2 Offline scenario

In the offline scenario, first, we construct data structure for all rules in rulelists. After that, we compare the construction and packet classification times of each method.

In Figure 4.5, we compare the packet classification of each method. But this time, we also take the average for all rulelists in each size. As expected, the proposed method has the best improvement (up to 28% better than the TupleMerge). The reason for getting these results is again using wider decision trees and fewer rules in each ruleset. In Figure 4.6, we compare the construction time with the PartitionSort method for the rulelists whose size is 64k while increasing the number of rule fields. We use only PartitionSort for this comparison because of the usage of decision trees as a data structure. As seen, our construction time is better (up to 88%) because the running time of our partitioning method is independent of the number of rule fields.

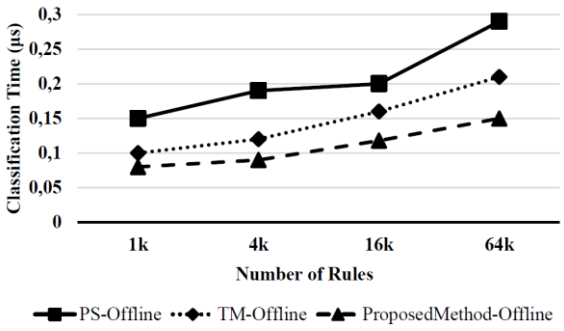


Figure 4.5: Offline Classification vs Rule Size.

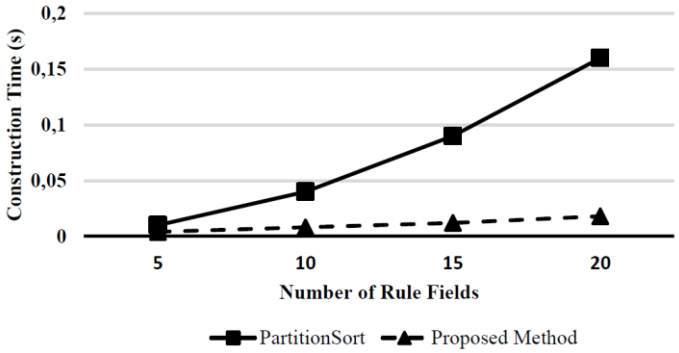


Figure 4.6: Construction Time vs Number of Rule Fields.

In Figure 4.7, we compare the packet classification time of our method with TupleMerge for the rulelists whose sizes are 64k while increasing the number of rule fields. As expected, our method is better up to 50% because TupleMerge has to search whole rule fields when it is necessary. And finally, we compare the packet classification time of our method in both online and offline scenarios in Figure 4.8. As seen, offline is better than online. But the difference is not so much because we don't consume so much time while inserting or deleting a rule from the data structure thanks to wider decision trees and fewer rules in each ruleset. Also, if there is a need to run the partitioning method, the running of it doesn't increase classification time much.

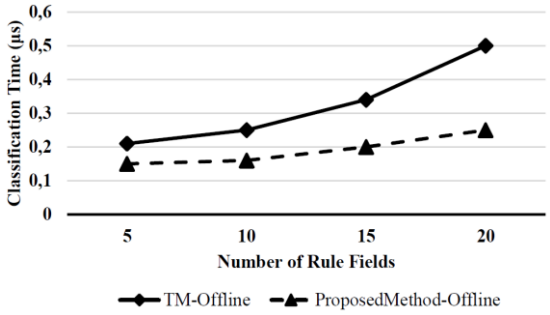


Figure 4.7: Offline Classification Time vs Number of Rule Fields.

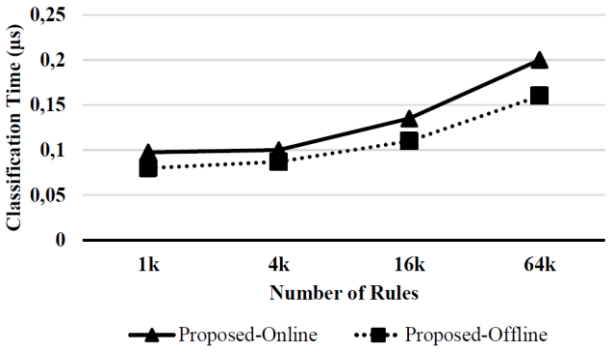


Figure 4.8: Online Classification vs Offline Classification.

5. CONCLUSIONS AND FUTURE WORK

Sofware-Defined Networking (SDN) brings a new, flexible, softwarization, and fast approach to the network management by separating the control plane and data plane from each other. But, it has its own problems like other technologies although it solves many problems. One of its problems that is investigated in this thesis is the congestion between the control plane and the data plane because of the centrality feature of the control plane. Another problem of SDN, which is also investigated in this thesis, is the slow packet classification and rule updating in OpenFlow vSwitches.

In this thesis, we firstly propose a fair and rapid Quality of Service (QoS) provisioning solution for the SDN controllers facing heterogeneous service flows in ultra-dense scenarios. We develop a novel flow-aware Management Engine to prevent congestions in the controller when heterogeneous URLLC, eMBB, and mMTC traffic suddenly increase. The fair processing of the proposed engine for heterogeneous flows provides faster response time to the incoming new packets (up to 53%). Also, as a result of faster response time, we decrease the e2e latency (up to 58%) and drop rates (up to 36%).

Secondly, we convert the rule partitioning problem to the interval partitioning problem and propose a classic greedy algorithm as a solution. As a result, we eliminate the rule replication problem in decision trees, and we make the running time of the partitioning solution independent from the number of rule fields while considering the characteristic of all rule fields. After that, we construct decision trees for each ruleset using the HyperCuts method and order all constructed data structures according to the highest priority value, which they have. Consequently, we decrease the construction time (up to 88%), packet classification time (up to 40% for online, up to 50% for an offline scenario with an increase in the number of the rule fields), and rule updating time (up to 15%).

As a future work, priority values for each flow type can be changed dynamically during the calculation of the newly defined priority value for each node of the tree structure for the congestion problem. In the thesis, it is assumed that these priority values of

flow types are static. On the other hand, relating these values to packet loss ratio or incoming amount of flow types and changing these values dynamically can indicate how the proposed solution affects performance metrics in different scenarios. Also, packet classification accuracy or errors can affect the e2e latency and packet drop rate of different flow types. To investigate the effects of these, labeling rules to show which flow types can be matched with these rules and comparing the classified packets at first with the matched and separately stored packets for each label can be used as a methodology for future work.

REFERENCES

[1] **Benzekki, K., El Fergougui, A., & Elbelrhiti Elalaoui, A.** (2016). Software-defined networking (SDN): a survey. *Security and communication networks*, *9*(18), 5803-5833.

[2] **Feamster, N., Rexford, J., & Zegura, E.** (2014). The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, *44*(2), 87-98.

[3] **McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., ... & Turner, J.** (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, *38*(2), 69-74.

[4] **Goransson, P., Black, C., & Culver, T.** (2016). *Software defined networks: a comprehensive approach*. Morgan Kaufmann.

[5] **Nygren, A., Pfaff, B., Lantz, B., Heller, B., Barker, C., Beckmann, C., ... & McDysan, D.** (2015). Openflow switch specification version 1.5. 1. *Open Networking Foundation, Tech. Rep.*

[6] **Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S.** (2008). NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, *38*(3), 105-110.

[7] **POX** <*www.noxrepo.org*>, date retrieved 29.05.2020

[8]**Beacon**<*https://openflow.stanford.edu/display/Beacon/Home*>, date retrieved 29.05.2020

[9] **Floodlight**<*https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller*>, date retrieved 29.05.2020

[10]**RYU<***www.ntt-review.jp/archive/ntttechnical.php?contents=ntr201408fa4.html* , date retrieved 29.05.2020

[11] **OpenDaylight** < *www.opendaylight.org* >, date retrieved 29.05.2020

[12] **ONOS** < *www.opennetworking.org/onos*/>, date retrieved 29.05.2020

[13] **Mijumbi, R., Serrat, J., Gorricho, J. L., Bouten, N., De Turck, F., & Boutaba, R.** (2015). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications surveys & tutorials*, *18*(1), 236-262.

[14] **Alvizu, R., Maier, G., Kukreja, N., Pattavina, A., Morro, R., Capello, A., & Cavazzoni, C.** (2017). Comprehensive survey on T-SDN: Software-defined networking for transport networks. *IEEE Communications Surveys & Tutorials*, *19*(4), 2232-2283.

[15] **Gupta, L., Jain, R., & Vaszkun, G.** (2015). Survey of important issues in UAV communication networks. *IEEE Communications Surveys & Tutorials*, *18*(2), 1123-1152.

[16] **Luo, H., Wu, K., Ruby, R., Liang, Y., Guo, Z., & Ni, L. M.** (2018). Software-defined architectures and technologies for underwater wireless sensor networks: A survey. *IEEE Communications Surveys & Tutorials*, *20*(4), 2855-2888.

[17] **Thyagaturu, A. S., Mercian, A., McGarry, M. P., Reisslein, M., & Kellerer, W.** (2016). Software defined optical networks (SDONs): A comprehensive survey. *IEEE Communications Surveys & Tutorials*, *18*(4), 2738-2786.

[18] **Baktir, A. C., Ozgovde, A., & Ersoy, C.** (2017). How can edge computing benefit from software-defined networking: A survey, use cases, and future directions. *IEEE Communications Surveys & Tutorials*, *19*(4), 2359-2391.

[19] **Zaidi, Z., Friderikos, V., Yousaf, Z., Fletcher, S., Dohler, M., & Aghvami, H.** (2018). Will SDN be part of 5G?. *IEEE Communications Surveys & Tutorials*, *20*(4), 3220-3258.

[20] **Das, T., Sridharan, V., Gurusamy, M.** (2020). A Survey on Controller Placement in SDN. *IEEE Communications Surveys & Tutorials*, *22*(1), 472-503.

[21] **Scott-Hayward, S., Natarajan, S., & Sezer, S.** (2015). A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, *18*(1), 623-654.

[22] **Nguyen, X. N., Saucez, D., Barakat, C., & Turletti, T.** (2015). Rules placement problem in OpenFlow networks: A survey. *IEEE Communications Surveys & Tutorials*, *18*(2), 1273-1286.

[23] **Yang, T., Liu, A. X., Shen, Y., Fu, Q., Li, D., & Li, X.** (2018, April). Fast openflow table lookup with fast update. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications* (pp. 2636-2644). IEEE.

[24] **Cisco.** (2018). Cisco Ultra 5G Packet Core Solution <*https://www.cisco.com*>, date retrieved 29.05.2020

[25] **Lien, S. Y., Hung, S. C., Deng, D. J., & Wang, Y. J.** (2017, December). Efficient ultra-reliable and low latency communications and massive machine-type communications in 5G new radio. In *GLOBECOM 2017-2017 IEEE Global Communications Conference* (pp. 1-7). IEEE.

[26] **Specifications, O. S.** (2015). 1.5. 1. *Open Networking Foundation*, *3*.

[27] **Daly, J., Bruschi, V., Linguaglossa, L., Pontarelli, S., Rossi, D., Tollet, J., ... & Yourtchenko, A.** (2019). Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking*, *27*(4), 1417-1431.

[28] **OpenFlow vSwitch** <*https://www.openvswitch.org*>, date retrieved 04.06.2020

[29] **Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., ... & Amidon, K.** (2015). The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (pp. 117-130).

[30] **Singh, S., Baboescu, F., Varghese, G., & Wang, J.** (2003, August). Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (pp. 213-224).

[31] **Wang, C., Liu, J., Li, B., Sohraby, K., & Hou, Y. T.** (2006). LRED: a robust and responsive AQM algorithm using packet loss ratio measurement. *IEEE Transactions on Parallel and Distributed Systems*, *18*(1), 29-43.

[32] **Cui, J., Lu, Q., Zhong, H., Tian, M., & Liu, L.** (2018). A load-balancing mechanism for distributed SDN control plane using response time. *IEEE Transactions on Network and Service Management*, *15*(4), 1197-1206.

[33] **Xu, H., Liu, J., Qian, C., Huang, H., & Qiao, C.** (2019). Reducing controller response time with hybrid routing in software defined networks. *Computer Networks*, *164*, 106891.

[34] **Wang, H., Xu, H., Huang, L., Wang, J., & Yang, X.** (2018). Load-balancing routing in software defined networks with multiple controllers. *Computer Networks*, *141*, 82-91.

[35] **Wang, T., Liu, F., & Xu, H.** (2017). An efficient online algorithm for dynamic SDN controller assignment in data center networks. *IEEE/ACM Transactions on Networking*, *25*(5), 2788-2801.

[36] **Sridharan, V., Mohan, P. M., & Gurusamy, M.** (2019). QoC-Aware Control Traffic Engineering in Software Defined Networks. *IEEE Transactions on Network and Service Management*.

[37] **Filali, A., Kobbane, A., Elmachkour, M., & Cherkaoui, S.** (2018, May). SDN controller assignment and load balancing with minimum quota of processing capacity. In *2018 IEEE International Conference on Communications (ICC)* (pp. 1-6). IEEE.

[38] **Filali, A., Cherkaoui, S., & Kobbane, A.** (2019, May). Prediction-Based Switch Migration Scheduling for SDN Load Balancing. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)* (pp. 1-6). IEEE.

[39] **Ruia, A., Casey, C. J., Saha, S., & Sprintson, A.** (2016, April). Flowcache: A cache-based approach for improving SDN scalability. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (pp. 610-615). IEEE.

[40] **Nayyer, A., Sharma, A. K., & Awasthi, L. K.** (2019). Laman: A supervisor controller based scalable framework for software defined networks. *Computer Networks*, *159*, 125-134.

[41] **Chang, D. Y., & Wang, P. C.** (2015). TCAM-based multi-match packet classification using multidimensional rule layering. *IEEE/ACM Transactions on Networking*, *24*(2), 1125-1138.

[42] **Bremler-Barr, A., Harchol, Y., Hay, D., & Hel-Or, Y.** (2018). Encoding short ranges in tcam without expansion: Efficient algorithm and applications. *IEEE/ACM Transactions on Networking*, *26*(2), 835-850.

[43] **Liu, A. X., Meiners, C. R., & Torng, E.** (2016). Packet classification using binary content addressable memory. *IEEE/ACM Transactions on Networking*, *24*(3), 1295-1307.

[44] **Qu, Y. R., & Prasanna, V. K.** (2015). High-performance and dynamically updatable packet classification engine on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, *27*(1), 197-209.

[45] **Hsieh, C. L., Weng, N., & Wei, W.** (2018). Scalable many-field packet classification for traffic steering in SDN switches. *IEEE Transactions on Network and Service Management*, *16*(1), 348-361.

[46] **Vamanan, B., Voskuilen, G., & Vijaykumar, T. N.** (2010). EffiCuts: optimizing packet classification for memory and throughput. *ACM SIGCOMM Computer Communication Review*, *40*(4), 207-218.

[47] **Qi, Y., Xu, L., Yang, B., Xue, Y., & Li, J.** (2009, April). Packet classification algorithms: From theory to practice. In *IEEE INFOCOM 2009* (pp. 648-656). IEEE.

[48] **He, P., Xie, G., Salamatian, K., & Mathy, L.** (2014, October). Meta-algorithms for software-based packet classification. In *2014 IEEE 22nd International Conference on Network Protocols* (pp. 308-319). IEEE.

[49] **Li, W., Li, X., Li, H., & Xie, G.** (2018, April). Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications* (pp. 2645-2653). IEEE.

[50] **Yingchareonthawornchai, S., Daly, J., Liu, A. X., & Torng, E.** (2018). A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking*, *26*(4), 1907-1920.

[51] **Srinivasan, V., Suri, S., & Varghese, G.** (1999, August). Packet classification using tuple space search. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (pp. 135-146).

[52] **Özçevik, M. E., Canberk, B., & Duong, T. Q.** (2017). End to end delay modeling of heterogeneous traffic flows in software defined 5G networks. *Ad Hoc Networks*, *60*, 26-39.

[53] **Kleinberg, J., & Tardos, E.** (2006). *Algorithm design*. Pearson Education India.

[54] **Taylor, D. E., & Turner, J. S.** (2007). Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking*, *15*(3), 499-511.

CURRICULUM VITAE

Name Surname **:** Mertkan AKKOÇ

Place and Date of Birth **:** EDİRNE / TURKEY, 05.10.1993

E-Mail **:** akkocm@itu.edu.tr

EDUCATION **:**

- **B.Sc.** **:** 2016, Istanbul Technical University, Electric Electronics Faculty, Electronics and Communication Engineering

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Akkoç, M., & Canberk B.** (2020, July). Flow-Aware QoS Engine for Ultra-Dense SDN Scenarios. In *2020 International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*
- **Akkoç, M., & Canberk B.** (2020, July). Interval Partitionig for Packet Classification in OpenFlow vSwitch, *IEEE Networking Letters*