Tiramisu: Fast and Scalable Traffic Splitting on Commodity Switches

Kun Huang[†], Qianpeng Su[‡]

[†]*Peng Cheng Laboratory*
Shenzhen, China
huangk03@pcl.ac.cn

[‡]*Southern University of Science and Technology*
Shenzhen, China
12032186@mail.sustech.edu.cn

***Abstract*—Traffic splitting is a key task for load balancing over multiple servers or paths. Load balancers often rely on commodity switches to implement traffic splitting within the ternary content addressable memory (TCAM). It is critical to reduce the amount of TCAMs allocated for this task since TCAMs are power-hungry and have limited capacities. Previous traffic splitting schemes have concentrated on minimizing the number of TCAM entries for a single service. However, these schemes suffer a great issue of scalability since a commodity switch with small-sized TCAMs needs to handle hundreds or thousands of services simultaneously. In this paper, we propose Tiramisu, an overlay approach to fast and scalable traffic splitting for multiple services on commodity switches. Tiramisu exploits wide SRAM words available in TCAMs to overlay multiple groups of rules within one same TCAM table. With rule overlaying, Tiramisu reduces the total number of TCAM entries allocated for multiple services. Experimental results on software simulations and P4 switches show that Tiramisu achieves significant reductions in number of TCAM entries by up to one order of magnitude for the imbalance error of 1% as well as achieves higher splitting throughput and faster rule updates compared to previous schemes.**

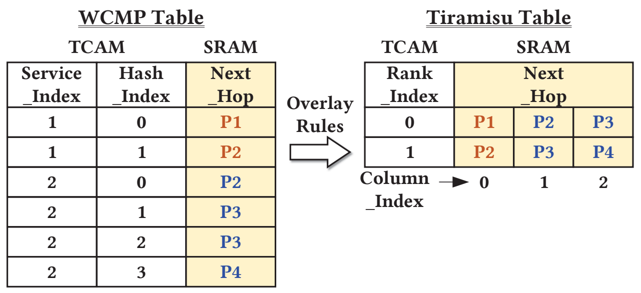
Keywords*—load balancing, traffic engineering, traffic splitting, ternary content addressable memory, programmable switch

I. Introduction

Traffic splitting is a key task in networks for load balancing over multiple servers or paths. For example, modern datacenters typically adopt network load balancers to spread traffic destined to a service across multiple servers and direct it to one of servers, achieving better performance, scalability, and reliability [1, 2, 3, 4, 5, 6, 26]. Dedicated network load balancers are expensive and lack the scalability and flexibility. Therefore, network operators increasingly rely on commodity switches to split the traffic load in proportion to the capacities of servers or paths [7, 8, 9, 10, 11, 12, 13, 14, 15, 27].

The ternary content addressable memory (TCAM) is used by commodity switches to implement traffic splitting [9, 10, 11, 12, 14, 15]. TCAMs have high-speed lookups due to their parallel search mechanism, where all the TCAM entries are searched in parallel. A TCAM entry stores a rule for a traffic splitting service which corresponds to a group of rules. However, these memories are power-hungry and have limited capacities on the order of a few thousand entries [11, 12, 14, 15]. Therefore, it is critical to reduce the amount of TCAMs allocated for traffic splitting on commodity switches.

This work was supported in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2019B151513000, the Major Key Project of PCL under Grant PCL2021A02, and the Guangdong Key-Area Research & Development Program under Grant 2020B010164001.



WCMP Table

TCAM		SRAM
Service _Index	Hash _Index	Next _Hop
1	0	P1
1	1	P2
2	0	P2
2	1	P3
2	2	P3
2	3	P4

Tiramisu Table

TCAM	SRAM		
Rank _Index	Next _Hop		
0	P1	P2	P3
1	P2	P3	P4
Column _Index	0	1	2

Fig. 1. Tiramisu reduces the number of TCAM entries allocated for two services over WCMP by overlaying two groups of rules within one TCAM table.

Several traffic splitting schemes [9, 10, 11, 12, 13, 14, 15] have been proposed to minimize the number of TCAM entries. Equal-cost multi-path (ECMP) [16, 17] is the most common approach to achieving a uniform traffic distribution by hashing the packet header. Weighted-cost multi-path (WCMP) [10, 13] is an extension of ECMP that supports a non-uniform traffic distribution by replicating entries in proportion to the weights of servers or paths. So WCMP requires prohibitively large amounts of TCAM entries. To overcome the drawback, Niagara [11, 12] trades off the accuracy for fewer entries by computing wildcard rules for an approximate traffic distribution. Sadeh et al. [14, 15] propose efficient algorithms to compute the minimal number of wildcard rules for a target approximate traffic distribution and to compute the optimal weight approximations for a fixed number of TCAM entries. Previous schemes concentrate on minimizing the number of TCAM entries for a single service. However, they suffer a great issue of scalability since a commodity switch with small-sized TCAMs needs to handle hundreds or thousands of services simultaneously. Recent studies [1, 2, 3, 4] have showed that a cloud datacenter hosts up to tens of thousands of services. Minimizing the number of TCAM entries allocated for a single service cannot scale well to large amounts of services. Therefore, it is desired to reduce the number of TCAM entries allocated for multiple services in a commodity switch.

In this paper, we propose Tiramisu, a novel overlay approach to fast and scalable traffic splitting for multiple services in small TCAMs. Tiramisu is a compact multi-service representation of

ISBN 978-3-903176-48-5© 2022 IFIP

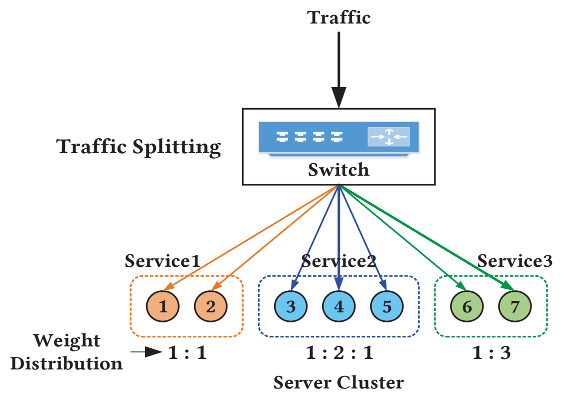
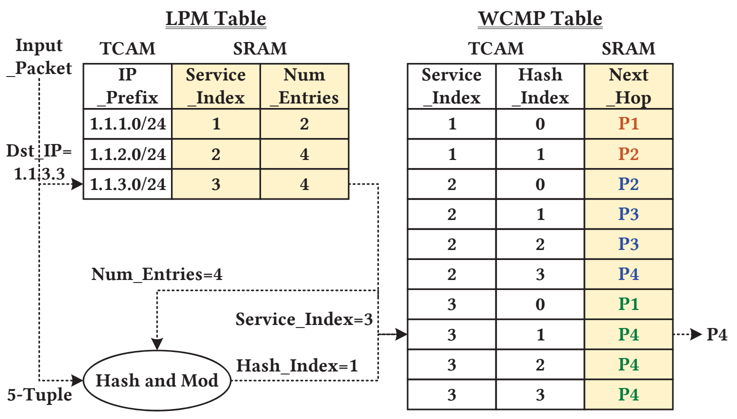


Fig. 2. Example of splitting traffic across three services over seven servers.



LPM Table

TCAM	SRAM	
IP _Prefix	Service _Index	Num _Entries
1.1.1.0/24	1	2
1.1.2.0/24	2	4
1.1.3.0/24	3	4

WCMP Table

TCAM		SRAM
Service _Index	Hash _Index	Next _Hop
1	0	P1
1	1	P2
2	0	P2
2	1	P3
2	2	P3
2	3	P4
3	0	P1
3	1	P4
3	2	P4
3	3	P4

Fig. 3. Illustration of traffic splitting with WCMP.

WCMP [10], scaling well to hundreds or thousands of services. Our goal is to reduce the number of TCAM entries allocated for multiple services while achieving fast traffic splitting and rule updates. To achieve this goal, Tiramisu exploits the technique of wide SRAM words [18, 19, 25] available in TCAMs to overlay multiple groups of rules for different services within one same TCAM table. Our design is motivated by the two observations: (1) different groups of rules share partial TCAM words; (2) one wide SRAM word can store multiple next hops of rules. These observations provide possible opportunities to encode multiple rules for different services into one TCAM entry. Fig. 1 shows an example of overlaying two groups of rules for two services 1 and 2 within one TCAM table. We see that the two groups of rules share partial hash indexes in TCAM words, and each wide SRAM word can store three next-hops of rules. As shown in Fig. 1, Tiramisu requires only two TCAM entries for two services, achieving 3x reductions over WCMP with six TCAM entries.

We conduct experiments on software simulations and P4 switches to evaluate the performance of Tiramisu and compare with previous schemes including ECMP [16], WCMP [10], and Niagara [11]. Our results demonstrate that Tiramisu has high performance and scalability. Specifically, the results on software simulations show that given the imbalance error of 1%, Tiramisu achieves up to one order of magnitude reductions in number of TCAM entries, improves 1.6x to 1.9x TCAM throughput, and reduces 1.7x rule update time compared to Niagara. In addition, the results on P4 switches show that given the imbalance error of 1%, Tiramisu-128 (with a 128-bit SRAM word for a TCAM entry) achieves 16x reductions in number of TCAM entries, improves 1.4x splitting throughput, and reduces 6.6x rule update time compared to WCMP.

The rest of this paper is organized as follows. We overview the background and related work on traffic splitting in Section II. Section III describes the design and analysis of Tiramisu. We present experimental results for Tiramisu evaluation in Section IV. Finally, Section V concludes this paper.

II. Background and Related Work

A. Traffic Splitting Background

Traffic splitting is critical for load balancing applications, such as cloud load balancing [1, 2, 3, 4, 5, 6], datacenter multi-path routing [7, 8, 20], and wide-area traffic engineering [21, 22]. Commodity switches use TCAMs to implement traffic splitting. A switch typically splits ingress traffic destined to a service over next-hop servers or paths according to the capacities of servers or paths. Fig. 2 shows a simple example of splitting traffic across three services over seven servers through a switch. The switch splits ingress traffic destined to a service according to the weight distribution. For service 1, the weight distribution is the ratio of 1:1, indicating that 1/2 traffic is sent to server 1 and another 1/2 traffic is sent to server 2. For service 2, the weight distribution is the ratio of 1:2:1, indicating that 1/4 traffic is sent to server 3, 2/4 traffic is sent to server 4, and another 1/4 traffic is sent to server 5. For service 3, the weight distribution is the ratio of 1:3, indicating that 1/4 traffic is sent to server 6 and 3/4 traffic is sent to server 7. We note that the weight distribution for a service is updated dynamically due to failures or changes in capacity.

Traffic splitting on commodity switches is required to satisfy three requirements. First, a traffic splitting scheme must scale to large amounts of services (e.g., tens of thousands of services in a modern datacenter [1, 2, 3, 4]) handled by a switch with small TCAMs. Second, the scheme must perform an accurate traffic split for minimizing traffic imbalance errors (i.e., more fractions of the traffic are sent to a server, which may overload the server and incur long latencies and service failures). Third, the scheme must quickly react to weight changes for maintaining the per-connection consistency and minimizing the traffic churn (i.e., a fraction of the traffic is reshuffled among different servers or paths). Therefore, a good traffic splitting scheme must satisfy these three requirements at the same time.

B. Related Work on Traffic Splitting

Over past years, several traffic splitting schemes [9, 10, 11, 12, 13, 14, 15, 16, 17] have been proposed for load balancing in commodity switches. These schemes fall into two categories: hash-based and match-based solutions.

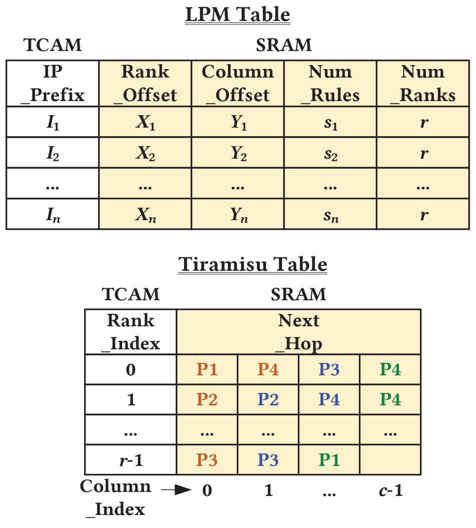
Hash-based traffic splitting: Hash-based solutions use hashing on specific fields of packet heads to determine the next-hop for each packet destined to a service. ECMP [16, 17] is the most popular hash-based traffic splitting scheme deployed in today's commodity switches. ECMP partitions the flow space of each service into a group of equal-sized entries each corresponding to a next-hop. ECMP uniformly splits traffic over a group of next-hops for a service by hashing the five-tuple of packet heads (i.e., the source IP, source port, destination IP, destination port, and protocol number). However, ECMP does not support a non-uniform traffic split as servers or paths have different capacities.

WCMP [10] is an extension of ECMP that supports non-uniform traffic splits. WCMP performs a weighted split by replicating entries in proportion to the weights of next-hops. The weight assigned to a next-hop is proportional to the capacity of a server or path associated with the next-hop. WCMP hashes the five-tuple of an incoming packet to a group of weighted entries for a service, and determines the next-hop (i.e., the outgoing port) for the packet. However, WCMP has high rule update overhead as it needs to update too many replicated entries for weight changes. DASH [13] improves the rule update performance over WCMP by comparing the hash to region boundaries.

Fig. 3 illustrates the process of traffic splitting with WCMP in a commodity switch. The switch maintains two TCAM tables: a LPM (Longest Prefix Matching) table and a WCMP table. The LPM table stores three prefix rules for three services 1, 2, and 3. Each entry in the LPM table contains the IP prefix (IP_Prefix), service index (Service_Index), and number of entries (Num_Entries) for a service stored in the WCMP table. The WCMP table stores three groups of next-hops for three services 1, 2, and 3. Service 1 has two entries associated with next-hops P1 and P2 due to the weight ratio of 1:1. Service 2 has four entries associated with next-hops P2, P3, P3, and P4 due to the weight ratio of 1:2:1. Service 3 has four entries associated with next-hops P1, P4, P4, and P4 due to the weight ratio of 1:3. On receiving an incoming packet with the destination IP (Dst_IP) 1.1.3.3, the switch first uses Dst_IP=1.1.3.3 as key to search the LPM table, and finds the matching entry pointing to service 3 (Service_Index=3) with four entries (Num_Entries=4) stored in the WCMP table. Then, the switch computes the hash index (Hash_Index=1) by hashing over the five-tuple of the packet head. Finally, the switch uses both Service_Index=3 and Hash_Index=1 as key to search the WCMP table, and finds the matching entry associated with the next-hop P4 (Next_Hop=P4) as the outgoing port.

However, it is hard to scale ECMP and WCMP to hundreds or thousands of services as commodity switches have limited capacities of TCAMs. For *n* services, ECMP and WCMP linearly allocate *n* groups of TCAM entries, leading to the space explosion in number of TCAM entries allocated for these services (e.g., *n*=10K services). Meanwhile, these TCAM entries cannot be shared among these different services since they have unique service indexes in the TCAM table (see Fig. 3). In this paper, we propose Tiramisu to reduce the total number of TCAM entries allocated for large amounts of services by sharing these TCAM entries among these services.

Match-based traffic splitting: Match-based solutions use wildcard matching on specific fields of incoming packet heads to determine the next-hop for each packet destined to a service. Wang et al. [9] propose a match-based traffic splitting scheme to approximate the weights with non-overlapping wildcard rules. Niagara [11] trades off the accuracy for fewer entries by using overlapping wildcard rules. Niagara [11] also takes the first step towards reducing the number of TCAM entries across services by packing multiple groups of rules into one TCAM table and sharing rules across services with similar weights. However, Niagara [11] does not significantly reduce the number of TCAM


LPM Table

TCAM	SRAM			
IP_Prefix	Rank_Offset	Column_Offset	Num_Rules	Num_Ranks
I_1	X_1	Y_1	s_1	r
I_2	X_2	Y_2	s_2	r
...	...	...	...	...
I_n	X_n	Y_n	s_n	r

Tiramisu Table

TCAM	SRAM			
Rank_Index	Next_Hop			
0	P1	P4	P3	P4
1	P2	P2	P4	P4
...	...	...	...	...
r-1	P3	P3	P1	
Column_Index →	0	1	...	c-1

Fig. 4. The architecture of Tiramisu with the LPM table and Tiramisu table.

entries allocated for large amounts of services. This is because it is very difficult to merge multiple wildcard rules across different services into one TCAM entry. Bit Matcher [14] is the optimal approach to efficiently computing the minimum number of wildcard rules for a single service given an approximate weight distribution. It is proved that Niagara [11] is the same optimal in number of TCAM entries as Bit Matcher [15]. In addition, Sadeh et al. [15] propose optimal weight approximations given a fixed number of TCAM entries. These above schemes concentrate on optimal representations or approximations for a single service but not for multiple services or across services.

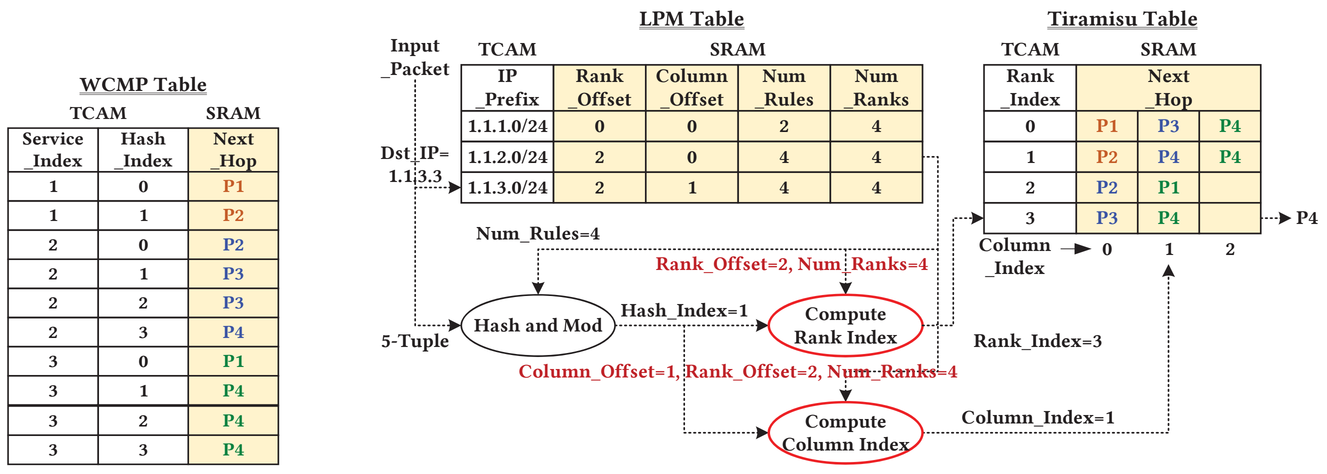
Load balancing applications: Traffic splitting schemes have been widely used in load balancing applications. Cloud load balancers (e.g., Ananta [1], Duet [2], Maglev [3], Beamer [5], and Cheetah [6]) use commodity switches and software load balancers for splitting traffic among thousands of services. CONGA [7] is a datacenter load balancing mechanism that splits traffic into flowlets (bursts of packets from a flow) among multiple paths, but it requires customized switch hardware support. HULA [8] uses programmable data planes for load balancing flowlets in datacenters. SilkRoad [4] uses switching ASICs to maintain per-connection states and consistency for datacenter load balancing. These works are orthogonal to our work, and Tiramisu can enhance these works by improving the performance and scalability of traffic splitting on switches.

III. Tiramisu for Traffic Splitting

We propose Tiramisu, a novel fast and scalable traffic splitting scheme for multiple services on commodity switches. Tiramisu aims to reduce the number of TCAM entries allocated for multiple services while sustaining high-speed traffic splitting and rule update performance. In this section, we first present the high-level overview of Tiramisu, and then describe the design of Tiramisu for both non-uniform and uniform traffic splitting. Finally, we analyze the space and time complexities of Tiramisu.

A. Tiramisu Overview

The basic idea behind Tiramisu is to overlay multiple groups of rules for different services within one same TCAM table. This



Service _Index	Hash _Index	Next _Hop
1	0	P1
1	1	P2
2	0	P2
2	1	P3
2	2	P3
2	3	P4
3	0	P1
3	1	P4
3	2	P4
3	3	P4

(a) WCMP.

IP _Prefix	Rank _Offset	Column _Offset	Num _Rules	Num _Ranks
1.1.1.0/24	0	0	2	4
1.1.2.0/24	2	0	4	4
1.1.3.0/24	2	1	4	4

Rank _Index	Next _Hop		
0	P1	P3	P4
1	P2	P4	P4
2	P2	P1	
3	P3	P4	
Column _Index	0	1	2

(b) Tiramisu.

Fig. 5. Illustration of non-uniform traffic splitting for three services with Tiramisu in comparision with WCMP.

design is motivated by the two observations as follows. First, different groups of rules for multiple services partially share the hash indexes in the TCAM word. The hash indexes are the hash values over the five-tuple of packet heads and have no logical meanings. For example, as shown in Fig. 5(a), three groups of rules for three services 1, 2, and 3 share the hash indexes (i.e., 0, 1, 2, 3). This provides an opportunity to merge multiple rules for different services into one TCAM entry with the same hash index. Second, TCAMs are augmented with wide SRAM words to improve the scalability and reduce the power consumption [18, 19, 25]. A usual SRAM word in a TCAM entry has the size of 32 or 64 bits. A wide SRAM word in a TCAM entry can be configured to have multiple usual words of size 128, 256, or 512 bits. This provide another opportunity to merge multiple next-hops of different rules into one wide SRAM word.

The architecture of Tiramisu consists of two TCAM tables: a LPM (Longest Prefix Matching) table and a Tiramisu table. Each entry in the LPM table stores the information of a service pointing to a group of rules for the service stored in the Tiramisu table. Each entry in the Tiramisu table stores the information of each group of rules for different services. Assume that a switch handles n services S_1, S_2, …, S_n using TCAMs, each service S_i with an IP prefix I_i and a group of s_i rules associated with s_i next-hops (e.g., P1, P2, P3, P4). Fig. 4 shows the architecture of Tiramisu with the LPM table and Tiramisu table. Each entry in the LPM table contains five parts: the IP prefix I_i (IP_Prefix) for a service i, rank offset X_i (Rank_Offset) for a group of rules for the service i stored in the Tiramisu table, column offset Y_i (Column_Offset) for a group of rules for the service i stored in the Tiramisu table, number s_i of rules (Num_Rules) for the service i stored in the Tiramisu table, and number r of ranks (Num_Ranks) in the Tiramisu table. We note that the IP prefix part is stored in the TCAM word, and other parts are stored in the SRAM word. The Tiramisu table is a matrix of rules with r ranks and c columns. Each next-hop of a rule is stored in a slot with the rank index (Rank_Index) and the column index (Column_Index). A group of rules for a service is stored in a column-oriented manner, where all next-hops of rules are stored contagiously in a column order.

From the high-level perspective, the LPM table is used as the index table that records the location information of each group of rules for multiple services. The Tiramisu table is used as the data table that stores each group of rules for multiple services in a column-oriented manner. The process of traffic splitting with Tiramisu is as follows. On receiving an incoming packet destined to a service, a commodity switch first uses the destination IP of the packet as key to search the LPM table, and determines the location information of a group of rules for the service that the packet belongs to. Then, the switch computes the rank index and column index of a rule that matches the packet by using the information of the service stored in the LPM table. Finally, the switch uses the rank index and column index to searches the Tiramisu table, and determines the matching rule associated with the next-hop port which the packet is forwarded to.

To search the Tiramisu table, we must first compute the rank index (Rank_Index) of a rule that matches a packet as:

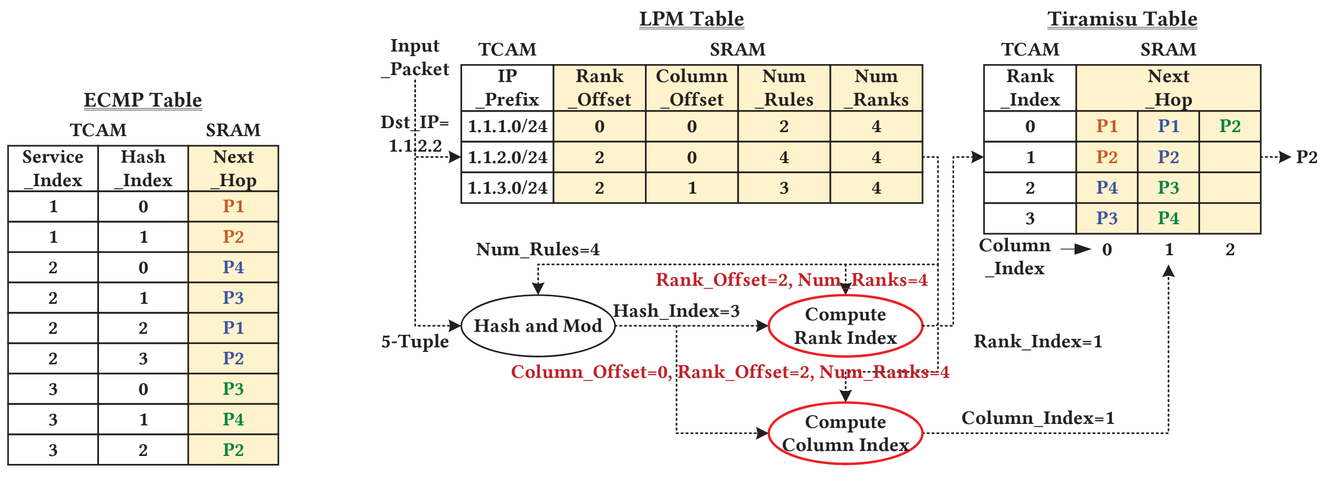
$$\begin{aligned} &Rank\_Index = \\ &(Hash\_Index + Rank\_Offset)\,\%\,Num\_Ranks \end{aligned} \tag{1}$$

where the hash index (Hash_Index) is computed by hashing over the five-tuple of the packet head and computing the hash modulo the number of rules (Num_Rules) for a service. Then, we must compute the column index (Column_Index) of a matching rule as:

$$\begin{aligned} &Column\_Index = Column\_Offset + \\ &(Hash\_Index + Rank\_Offset) \,/\, Num\_Ranks \end{aligned} \tag{2}$$

The rank index (Rank_Index), column index (Column_Index), and number of ranks (Num_Ranks) are extracted from a TCAM entry pointing to a service of a packet in the LPM table.

Since TCAMs are a scarce resource with limited capacities, Tiramisu is a compact multi-service representation of WCMP that minimizes the number of TCAM entries (i.e., the number of rules) allocated for multiple services given an approximate weight distribution. We use a weight reduction algorithm called weight approximation with minimum rules (WAMR) proposed



ECMP Table

Service_Index (TCAM)	Hash_Index (TCAM)	Next_Hop (SRAM)
1	0	P1
1	1	P2
2	0	P4
2	1	P3
2	2	P1
2	3	P2
3	0	P3
3	1	P4
3	2	P2

LPM Table

IP_Prefix (TCAM)	Rank_Offset (SRAM)	Column_Offset (SRAM)	Num_Rules (SRAM)	Num_Ranks (SRAM)
1.1.1.0/24	0	0	2	4
1.1.2.0/24	2	0	4	4
1.1.3.0/24	2	1	3	4

Tiramisu Table

Rank_Index (TCAM)	Next_Hop (SRAM) Column_Index 0	1	2
0	P1	P1	P2
1	P2	P2	
2	P4	P3	
3	P3	P4	

(a) ECMP. (b) Tiramisu.

Fig. 6. Illustration of uniform traffic splitting for three services with Tiramisu in comparison with ECMP.

in [10] to minimize the number of rules for each service given a maximum imbalance error of weight approximations. Moreover, we use an efficient greedy solution proposed in [10] to WAMR. This solution starts from the smallest set of rules each with the weight of 1, and then greedily searches for the next smaller set of rules with the least ratio of the maximum imbalance error.

B. *Non-Uniform and Uniform Traffic Splitting*

Tiramisu supports non-uniform and uniform traffic splitting for multiple services on commodity switches as follows.

Fig. 5 shows the process of non-uniform traffic splitting for multiple services with Tiramisu in comparison with WCMP. As shown in Fig. 5(a), the WCMP table contains three groups of rules for three services 1, 2, and 3 with the weight ratio of 1:1, 1:2:1, and 1:3, respectively. We use rule overlaying to build the Tiramisu table with four ranks and three columns, as shown in Fig. 5(b). We also build the LPM table to record the information of each group of rules for three services stored in the Tiramisu table. On receiving an incoming packet with the destination IP 1.1.3.3, the switch first uses Dst_IP=1.1.3.3 to search the LPM table, and finds the matching entry with the IP prefix 1.1.3.0/24. Then, the switch uses the five-tuple of the packet head to compute the hash modulo the number of ranks (Num_Ranks=4) in the Tiramisu table, outputting the hash index (Hash_Index=1). Next, the switch computes the rank index (Rank_Index=3) by (1) and the column offset (Column_Offset=1) by (2), with the rank offset (Rank_Offset=2) and column offset (Column_Offset=1). Finally, the switch uses the rank index and column index as key to search the Tiramisu table, and determines the matching entry associated with the next-hop (Next_Hop=P4) for the packet.

Tiramisu has the process of uniform traffic splitting similar to that of non-uniform traffic splitting. Fig. 6 shows the process of uniform traffic splitting for multiple services with Tiramisu in comparison with ECMP. As shown in Fig. 6(a), the ECMP table contains three groups of rules for three services 1, 2, and 3 each with the uniform weight distribution. On receiving an incoming packet with the destination IP 1.1.2.2, as shown in Fig. 6(b), the switch first uses Dst_IP=1.1.2.2 to search the LPM table, and finds the matching entry with the IP prefix 1.1.2.0/24. Then, the switch computes the hash index (Hash_Index=3) by hashing over the five-tuple of the packet head. Next, the switch computes the rank index (Rank_Index=1) by (1) and the column index (Column_Index=1) by (2). Finally, the switch uses the rank index and column index as key to search the Tiramisu table, and determines the matching entry associated with the next-hop (Next_Hop=P2) for the packet.

C. *Rule Updates*

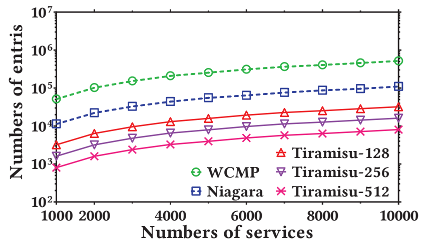
Tiramisu needs to update rules when weights change due to next-hop capacity changes. Tiramisu aims to efficiently update rules in TCAMs while minimizing traffic churn (i.e., packets of the traffic are reshuffled among different servers due to weight updates). Tiramisu has two rule update strategies. When weights change frequently, Tiramisu computes new rules from old rules and then relocates the affected rules by next-hop moving. When weights change infrequently, Tiramisu computes a group of new rules for a service from scratch in the controller and then installs them into a data-plane switch. Since weights update infrequently in most practical applications [11, 12, 14, 15], we adopt the latter strategy to recompute new rules from scratch for weight updates in the controller, and then reinstall them into a data-plane switch. Our experiments in Section IV show that Tiramisu outperforms previous schemes in terms of rule update time.

D. *Space and Time Complexities Analysis*

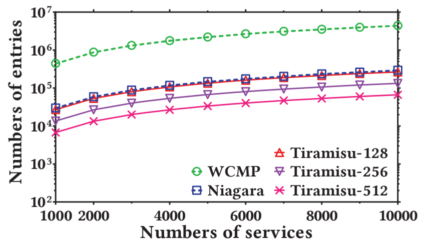
We first analyze the space complexity of Tiramisu in terms of the number of TCAM entries. Given n services S_1, S_2, …, S_n, Tiramisu uses the WAMR algorithm [10] to compute a group of s_i rules for a service S_i. Therefore, there are totally s_1+s_2…+s_n rules each with a next-hop of h bits (i.e., 2^h ports in a switch). Assumed that the Tiramisu table has r ranks and c columns, we compute the total number of TCAM entries E of both the LPM table and Tiramisu table as:

$$E = n + r \ge n + \sum_{i=1}^{n} s_i / c \qquad (3)$$

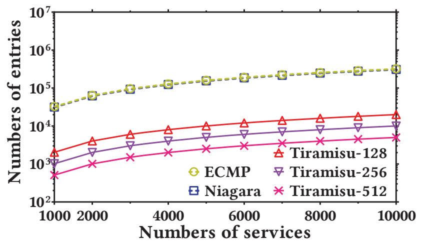
where the LPM table has n TCAM entries and the Tiramisu table has r TCAM entries. Eq. (3) shows that the total number E of TCAM entries allocated for n services decreases as the number



(a) Imbalance error 1%

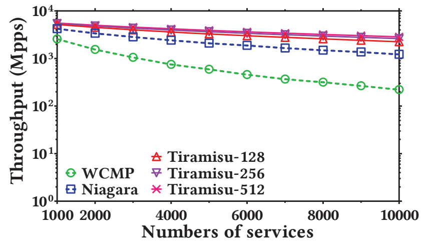


(b) Imbalance error 0.1%

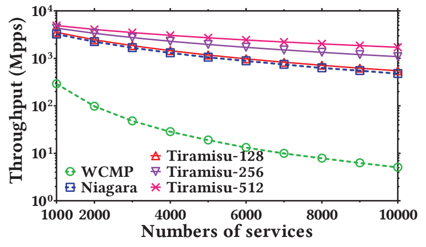


(c) Uniform traffic splitting

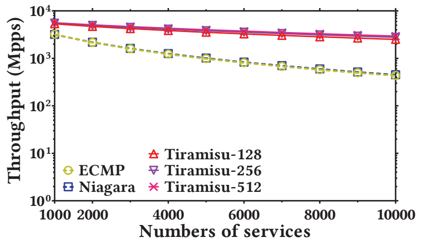
Fig. 7. The number of TCAM entries required for non-uniform and uniform traffic splitting.



(a) Imbalance error 1%



(b) Imbalance error 0.1%



(c) Uniform traffic splitting

Fig. 8. TCAM throughput in Mpps achieved for non-uniform and uniform traffic splitting.

c of columns in the Tiramisu table increases (i.e., the size of a wide SRAM word in the TCAM table increases).

Next, we analyze the time complexity of Tiramisu for traffic splitting and rule updates. The splitting time T_{split} is the search time of both the LPM table and Tiramisu table in a TCAM. The search time T of a TCAM table consists of the search time t_{TCAM} of TCAM words and the access time t_{SRAM} of SRAM words. We compute the splitting time T_{split} of Tiramisu as:

$$\begin{aligned} T_{\text{split}} &= T^{\text{LPM}} + T^{\text{Tiramisu}} \\ &= (t_{\text{TCAM}}^{\text{LPM}} + t_{\text{SRAM}}^{\text{LPM}}) + (t_{\text{TCAM}}^{\text{Tiramisu}} + t_{\text{SRAM}}^{\text{Tiramisu}}) \end{aligned} \tag{4}$$

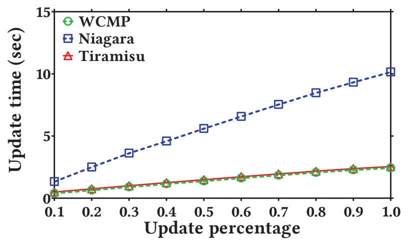
Eq. (4) shows that Tiramisu has different T^{LPM} and T^{Tiramisu}, compared to previous schemes such as WCMP and ECMP. The update time T_{update} consists of the rule computation time T_{compute} in the controller and the rule installation time T_{install} in the switch. We compute the update time T_{update} of Tiramisu as:

$$T_{\text{update}} = T_{\text{compute}} + T_{\text{install}} \tag{5}$$

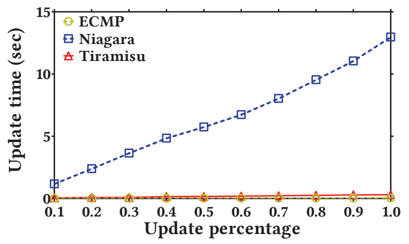
Our experiments in Section IV demonstrate the time and space complexities of Tiramisu, and show that Tiramisu reduces the number of TCAM entries by up to one order of magnitude as well as improves traffic splitting and rule update performance compared to previous schemes.

IV. Evaluation

In this section, we evaluate the performance of Tiramisu and compare with previous schemes including ECMP [16], WCMP [10], and Niagara [11]. We do not compete with other schemes [12, 14]. This is because Niagara [11] is proved to be the optimal representation with the minimum number of TCAM entries for a single service given a desired approximate weight distribution [14]. We conduct experiments on software simulations and P4 switches to validate the performance and scalability of Tiramisu.



(a) Non-uniform traffic splitting

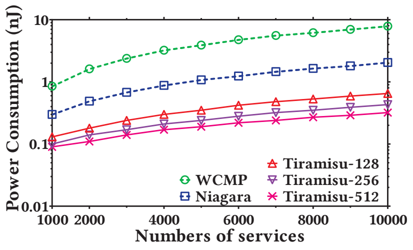


(b) Uniform traffic splitting

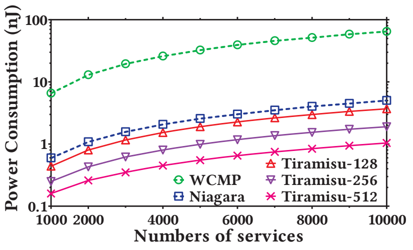
Fig. 9. Update time for reconstructing all rules in the controller.

A. *Experiments on Software Simulations*

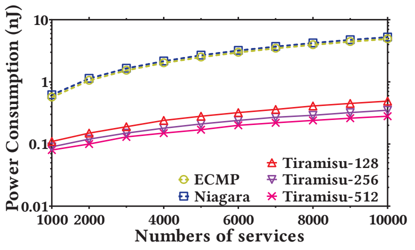
We implement Tiramisu and previous schemes in software, and conduct simulation experiments to measure the performance metrics for given weight approximations. These metrics include the number of TCAM entries, the TCAM throughput, the update time for reconstructing all rules in the controller, and the TCAM power consumption. In the experiments, we set the imbalance error for a weight approximation to no more than 1% or 0.1%,



(a) Imbalance error 1%



(b) Imbalance error 0.1%



(c) Uniform traffic splitting

Fig. 10. TCAM power consumption required for non-uniform and uniform traffic splitting.

and synthesize 1000 to 10000 services for traffic splitting on a commodity switch with up to 256 next-hop ports. For a service, we generate the number of next-hop ports to follow the binomial distribution [11], and generate the weight of a next-hop port to follow the Gaussian distribution [11]. In addition, we set the size of a next-hop port to 8 bits for up to 256 switch ports, and set the size of a wide SRAM word in a TCAM entry to 128, 256, or 512 bits. Moreover, in order to evaluate the TCAM throughput and TCAM power consumption, we exploit an open-source TCAM simulator [23], where the TCAM chip is manufactured with a 28*nm* processor to estimate the latency of a single TCAM table lookup. We run the experiments on a server with Intel Core i7-9700 CPU@3.0GHZ (12MB L3 cache) and 16GB DDR4 main memory. The results are averaged over ten trials.

Fig. 7 shows the number of TCAM entries required for non-uniform and uniform traffic splitting. Fig. 7(a) and Fig. 7(b) compare the number of TCAM entries for two imbalance errors of 1% and 0.1%, respectively. We see that Tiramisu achieves significant reductions in number of TCAM entries by up to one order of magnitude compared to previous schemes. Specifically, compared to WCMP, Tiramisu-128 (with a 128-bit SRM word), Tiramisu-256 (with a 256-bit SRM word), and Tiramisu-512 (with a 512-bit SRM word) separately reduce the number of TCAM entries by 16x, 32x, and 64x for both two imbalance errors. Compared to Niagara, Tiramisu-128, Tiramisu-256, and Tiramisu-512 separately reduce the number of TCAM entries by 3.4x, 6.9x, and 13.8x for the imbalance error of 1% (see Fig. 7(a)) and by 1.1x, 2.2x, and 4.4x for the imbalance error of 0.1% (see Fig. 7(b)). In addition, we see that the number of TCAM entries required by Tiramisu decreases as the size of a wide SRAM word increases from 128 to 512 bits and as the imbalance error increases from 0.1% to 1%. This is because Tiramisu leverages wider SRAM words for fewer TCAM entries, and requires more TCAM entries for higher accuracy.

Fig. 7(c) shows the number of TCAM entries required for uniform traffic splitting. We also see that Tiramisu achieves up to one order of magnitude reductions in the number of TCAM entries compared to previous schemes. Specifically, Tiramisu-128, Tiramisu-256, and Tiramisu-512 reduce the number of TCAM entries by 16x, 32x, and 64x compared to ECMP and Niagara. We note that both Niagara and ECMP require the same number of TCAM entries allocated for thousands of services. This is because Niagara regresses to ECMP in the scenario of uniform traffic splitting.

Fig. 8 shows the TCAM throughput in million packets per second (Mpps) for non-uniform and uniform traffic splitting. We see that Tiramisu achieves higher TCAM throughput than previous schemes. Specifically, as shown in Fig. 8(a), Tiramisu improves the TCAM throughput by 6.0x to 7.3x and by 1.6x to 1.9x compared to WCMP and Niagara for the imbalance error of 1%, respectively. As shown in Fig. 8(b), Tiramisu improves the TCAM throughput by 64.1x to 167.1x and by 1.1x to 2.6x compared to WCMP and Niagara for the imbalance error of 0.1%, respectively. In addition, compared to ECMP and Niagara, Tiramisu improves the TCAM throughput by 3.7x to 4.2x for uniform traffic splitting, as shown in Fig. 8(c). Note that Niagara achieves the same TCAM throughput as ECMP for uniform traffic splitting because Niagara regresses to ECMP in the case. Eq. (4) shows that the total splitting throughput is composed of both the TCAM throughput and SRAM throughput. Therefore, we will compare the total splitting throughput of Tiramisu and previous schemes on P4 switches in the next subsection.

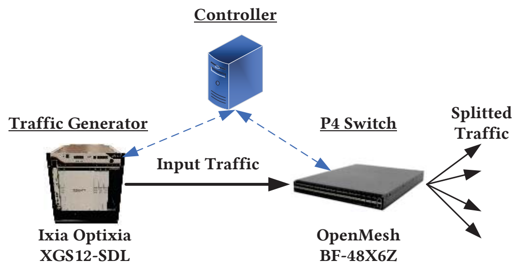
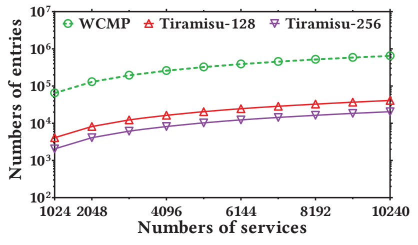


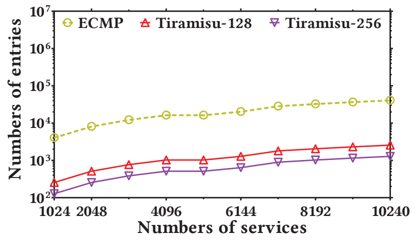
Fig. 11. A topology for traffic splitting on a P4 switch.

Fig. 9 shows the update time of reconstructing all rules in the controller with different percentages of updated services. We see that Tiramisu almost requires the same update time as both WCMP and ECMP. This is because Tiramisu is a compact multi-service representation of WCMP and ECMP, without additional update overhead for computing exact-match rules. In addition, compared to Niagara, Tiramisu reduces the rule update time by 1.7x for non-uniform traffic splitting (see Fig. 9(a)) and by 2.0x for uniform traffic splitting (see Fig. 9(b)). This is because Tiramisu requires fewer update time for computing exact-match rules than Niagara for computing wildcard rules.

Fig. 10 shows the TCAM power consumption required for non-uniform and uniform traffic splitting. We see that Tiramisu requires lower TCAM power consumption than previous schemes. This is because the number of TCAM entries for a search dominates the TCAM power consumption, and Tiramisu requires fewer TCAM entries than previous schemes (see Fig. 7). Specifically, as shown in Fig. 10(a), Tiramisu reduces the

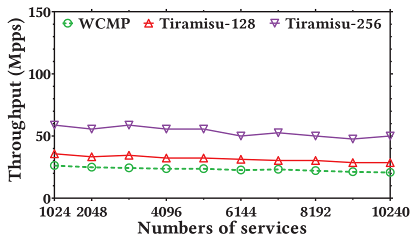


(a) Non-uniform traffic splitting

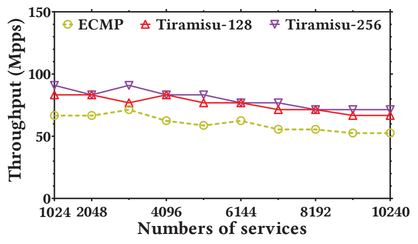


(b) Uniform traffic splitting

Fig. 12. The number of TCAM entris on a P4 switch.



(a) Non-uniform traffic splitting



(b) Uniform traffic splitting

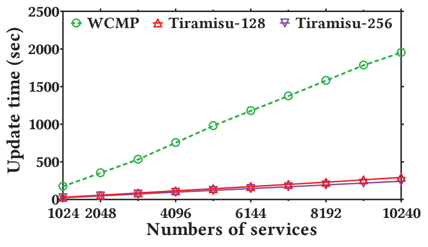
Fig. 13. Splitting throughput in Mpps on a P4 switch.

TCAM power consumption by 10.6x to 19.8x and by 2.9x to 5.4x compared to WCMP and Niagara for the imbalance error of 1%, respectively. As shown in Fig. 10(b), Tiramisu reduces the TCAM power consumption by 17.0x to 57.3x and by 1.4x to 4.6x compared to WCMP and Niagara, respectively. In addition, Tiramisu reduces the TCAM power consumption by 9.3x to 14.9x compared to both ECMP and Niagara for uniform traffic splitting, as shown in Fig. 10(c). We note that Niagara requires the same TCAM power consumption as ECMP in the scenario of uniform traffic splitting.

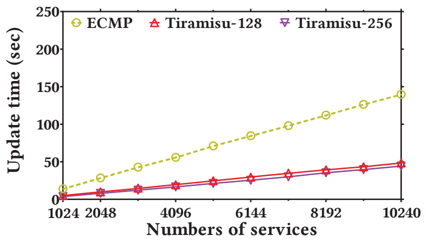
B. *Experiments on P4 Switches*

We implement Tiramisu, ECMP, and WCMP in P4 [24, 25], and conduct experiments to evaluate the performance using a OpenMesh BF-48X6Z P4 switch with Barefoot Tofino 3.2Tbps. We do not implement and evaluate Niagara [11] in P4 because the OpenMesh BF-48X6Z P4 switch only compiles prefix rules but not suffix rules generated by Niagara [11]. We use an Ixia Optixia XGS12-SDL test instrument to generate 10Gbps traffic through a OpenMesh BF-48X6Z P4 switch with up to 4 next-hop ports as shown in Fig. 11. In the experiments, we synthesize 1K to 10K services for the target imbalance error of 1%. For a service, we also generate the number of next-hop ports to follow the binomial distribution [11], and generate the weight of a next-hop port to follow the Gaussian distribution [11]. In addition, we set the size of a next-hop port to 8 bits, and set the size of a wide SRAM word to 128 or 256 bits. We measure the performance metrics including the number of TCAM entries, the splitting throughput, and the rule update time for compiling and installing all rules in P4.

Fig. 12 shows the number of TCAM entries on a P4 switch. We see that Tiramisu achieves significant reductions in number of TCAM entries by up to one order of magnitude compared to previous schemes. Specifically, Tiramisu-128 and Tiramisu-256 separately reduce the number of TCAM entries by 16x and 32x compared to both WCMP and ECMP. This is because Tiramisu



(a) Non-uniform traffic splitting



(b) Uniform traffic splitting

Fig. 14. Update time for compiling and installing all rules on a P4 switch.

encodes 16 or 32 next-hops of rules into one TCAM entry as the wide SRAM word is of size 128 or 256 bits and the next-hop is of size 8 bits.

Fig. 13 shows the splitting throughput in million packets per second (Mpps) on a P4 switch. We see that Tiramisu achieves higher splitting throughput than previous schemes. Specifically, as shown in Fig. 13(a), Tiramisu-128 and Tiramisu-256 separately improve non-uniform splitting throughput by 1.4x and 2.3x compared to WCMP. As shown in Fig. 13(b), Tiramisu-128 and Tiramisu-256 almost have the same uniform splitting throughput, and improve 1.3x throughput over ECMP.

Fig. 14 shows the update time for compiling and installing all rules on a P4 switch. We see that Tiramisu achieves faster rule updates than previous schemes. Specifically, as shown in Fig. 14(a), Tiramisu-128 and Tiramisu-256 separately reduce the update time by 6.6x and 7.9x compared to WCMP for non-uniform traffic splitting. As shown in Fig. 14(b), Tiramisu-128 and Tiramisu-256 separately reduce the update time by 2.9x and 3.4x compared to ECMP for uniform traffic splitting. In addition, the update time of Tiramisu increases gracefully as the number of services supported by a P4 switch increases.

V. Conclusion

This paper presents Tiramisu, a simple yet effective overlay approach to fast and scalable multi-service traffic splitting on commodity switches. Tiramisu aims to reduce the number of TCAM entries allocated for multiple services while achieving high splitting throughput and fast rule updates. Tiramisu exploits wide SRAM words to overlay multiple groups of rules into one same TCAM table. Experiments on software simulations and P4 switches are conducted to evaluate Tiramisu and compare with previous traffic splitting schemes. The results demonstrate that Tiramisu has high performance and scalability, and outperforms previous schemes in space and time efficiency. Specifically, the results on software simulations show that given the imbalance error of 1%, Tiramisu reduces the number of TCAM entries by up to one order of magnitude, improves the TCAM throughput by up to 1.9x, and reduces the rule update time by 1.7x compared to Niagara. In addition, the results on P4 switches show that given the imbalance error of 1%, Tiramisu-128 (with a 128-bit SRAM word in a TCAM entry) reduces the number of TCAM entries by 16x, improves the splitting throughput by 1.4x, and reduces the rule update time by 6.6x compared to WCMP.

References

[1] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, “Ananta: cloud scale load balancing,” in ACM SIGCOMM, 2013.

[2] R. Gandhi, H. Liu, Y. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: cloud scale load balancing with hardware and software," in ACM SIGCOMM, 2014.

[3] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in USENIX NSDI, 2016.

[4] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: making stateful layer-4 load balancing fast and cheap using switching asics," in ACM SIGCOMM, 2017.

[5] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in USENIX NSDI, 2018.

[6] T. Barbette, C. Tang, H. Yao, D. Kostic, G. Maguire, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency," in USENIX NSDI, 2020.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: distributed congestion-aware load balancing for datacenters," in ACM SIGCOMM, 2014.

[8] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: scalable load balancing using programmable data planes," in ACM SOSR, 2016.

[9] R. Wang, D. Butnariu, and J. Rexford, “Openflow-based server load balancing gone wild,” in USENIX Hot-ICE, 2011.

[10] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: weighted cost multipathing for improved fairness in data centers," in ACM EuroSys, 2014.

[11] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in ACM CoNEXT, 2015.

[12] O. Rottenstreich, Y. Kanizo, H. Kaplan, and J. Rexford, “Accurate traffic splitting on commodity switches,” in ACM SPAA, 2018.

[13] K.-F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Adaptive weighted traffic splitting in programmable data planes," in ACM SOSR, 2020.

[14] Y. Sadeh, O. Rottenstreich, A. Barkan, Y. Kanizo, and H. Kaplan, "Optimal representations of a traffic distribution in switch memories," IEEE/ACM Transactions on Networking, 28 (2): 930-943, 2020.

[15] Y. Sadeh, O. Rottenstreich, and H. Kaplan, "Optimal approximations for traffic distribution in bounded switch memories," in ACM CoNext, 2020.

[16] C. Hopps, "Analysis of an equal-cost multi-path algorithm," RFC 2992, 2000.

[17] M. Chiesa, G. Kindler, and M. Schapira, “Traffic engineering with equal-cost-multipath: an algorithmic perspective,” IEEE/ACM Transactions on Networking, 25(2): 779-792, 2017.

[18] W. Lu and S. Sahni, “Low power tcams for very large forwarding tables,” in IEEE INFOCOM, 2008.

[19] T. Banerjee, S. Sahni, and G. Seetharaman, "PC-TRIO: a power efficient tcam architecture for packet classifiers," IEEE Transactions on Computers, 64(4): 1104-1118, 2015.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in ACM SIGCOMM, 2008.

[21] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven wan," in ACM SIGCOMM, 2013.

[22] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat, “B4: experience with a globally-deployed software defined wan,” in ACM SIGCOMM, 2013.

[23] B. Agrawal and T. Sherwood, “Ternary cam power and delay model: extensions and uses,” IEEE Transactions on VLSI, 16(5): 554-564, 2008.

[24] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," ACM Computer Communication Review, 44(3): 87–95, 2014.

[25] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," in ACM SIGCOMM, 2013.

[26] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu, E. Song, J. Zhang, T. Huang, and S. Zhu, “Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches,” in ACM SIGCOMM, 2021.

[27] Y. Sadeh, O. Rottenstreich, and H. Kaplan, "How much tcam do we need for splitting traffic," in ACM SOSR, 2021.